VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nové možnosti Unreal Enginu 4

Possibilities of Unreal Engine 4

2017 Martin Mada

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Mada**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Nové možnosti Unreal Enginu 4
Possibilities of Unreal Engine 4

Jazyk vypracování: čeština

Zásady pro vypracování:

Nová verze Unreal Enginu 4 sebou přináší pokročilé možnosti v oblasti vývoje her, vzdělávacích aplikací, vizualizací apod. To je velmi důležité zejména při simulací jevů a procesů, které nelze v reálnem světě jednoduše demonstrovat. Důvodem může být například to, že je to technicky nemožné nebo by přitom mohlo dojít k ohrožení zdraví skupiny nebo jedinců. Proto již dnes existují např. simulace operací v lékařství nebo simulace na úrovní molekulárního světa. Cílem této práce je zaměřit se na možnosti vizualizace v novém Unreal Engine 4. Práce se bude zabývat popisem a demonstrací vybraných části tohoto enginu a výsledky budou použity v aplikaci pro simulování reálného provozu virtuální jaderné elektrárny.

1. Nastudujte možnosti a techniky při návrhu a implementaci projektu pro realistickou vizualizaci v Unreal Engine 4.
2. V práci se zaměřte zejména na následující části:
a) Hra více hráčů (multiplayer).
b) Komunikace více hráčů.
c) Inventář.
d) Ovládání objektů.
3. Teoretické znalosti využijte k implementaci ukázkových příkladů pro zpracovávaná témata.
4. Příklady vhodně zakomponujte po domluvě s vedoucím do projektu pro vizualizaci jaderné elektrárny, popřípadě jiných vhodných projektů.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. Dubna 2017

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. Dubna 2017

Abstrakt

Nová verze Unreal Enginu 4 sebou přináší pokročilé možnosti v oblasti vývoje her, vzdělávacích aplikací, vizualizací apod. To je velmi důležité zejména při simulací jevů a procesů, které nelze v reálnem světě jednoduše demonstrovat. Důvodem může být například to, že je to technicky nemožné nebo by přitom mohlo dojít k ohrožení zdraví skupiny nebo jedinců. Proto již dnes existují např. simulace operací v lékařství nebo simulace na úrovní molekulárního světa. Cílem této práce je zaměřit se na možnosti vizualizace v novém Unreal Engine 4. Práce se zabývá popisem a demonstrací vybraných části tohoto enginu a výsledky jsou následně použity ve hře pro zkoumání chování lidí v bludišti.

Klíčová slova: Unreal Engine, engine

Abstract

The new version of Unreal Engine 4 brings some advanced possibilities in game development, educational applications, visualizations etc. This is very important especially during simulations of events and processes, which we are not able to demonstrate in real life. The reason can be, that it is technically impossible or it can cause a serious health threat for individual or a group of people. Therefore nowdays there exist for example surgery simulations or simulations of molecular world. The aim of this work is to focus on possibilities of visualizations in new Unreal Engine 4. This work describes and demonstrate chosen parts of this engine and it's results are used in game Bludiště, which was made for exploration of human behavior in maze.

Key Words: Unreal Engine, engine

Obsah

Seznam použitých zkratek a symbolů

UE4	–	Unreal Engine 4
RPG	–	Role-playing Game
MMORPG	–	Massively Multiplayer Online Role-playing Game
LAN	–	Local Area Network

Seznam obrázků

1 Úvod

Je určitě nepopíratelným faktem, že se v dnešní době vyvíjí stále více počítačových her a simulací a tudíž rostou i nároky na ně. Všechny společnosti se předbíhají v tom, která vydá kvalitněji zpracovaný produkt. Mnoho těchto společností využívá pro tvorbu svých aplikací již existující vývojová prostředí neboli enginy, jelikož jejich vývoj je jak časově, tak finančně náročný a navíc enginy již existující jsou již prověřeny řadou předchozích aplikací, které v nich byly vytvořeny a nabízí velice kvalitní **real time rendering**. V době psaní této práce je na trhu celá řada enginů, ovšem nejvyužívanějšími jsou bezpochyby Unity, Unreal Engine 4 a Cry Engine.

Výše zmíněné enginy dobyly herní průmysl především díky tomu, že jsou k dispozici zcela zdarma ke stažení a poskytují také náhled do svého kódu, což je dostalo mezi širší okruh lidí a ne pouze k velkým firmám, které si jejich používání mohou zaplatit. Je ovšem důležité zmínit, že všechny výše zmíněné enginy si nárokují podíl na zisku z dané aplikace, pokud dosáhne určitého zisku.

Všechny tyto enginy jsou také multiplatformní a neslouží pouze k vývoji aplikací pro počítače ale také pro herní konzole a mobilní zařízení. Právě ve vývoji mobilních aplikací se v dnešní době nejvíce uplatňuje engine Unity , který se v této oblasti stal velice spolehlivým a užitečným nástrojem. Více informací o tomto enginu je možné nalézt zde [1]. Naproti tomu Cry Engine se plně soustředí na vývoj počítačových her, kde nabízí plnou podporu nejmodernějších technologií, ke kterým se v rámci jeho nové verze Cry Engine 5 přidala také podpora virtuální reality [2].

Cílem této práce je popis zadaných částí posledního z trojice zmíněných enginů a sice Unreal Enginu 4. Tento engine byl vyvíjen od roku 1998 společností Epic Games. Z původního konceptu, který byl určen pouze pro vývoj střílečeк z pohledu první osoby, se vyvinul v multiplatformní vývojové prostředí. Jeho jádro je napsáno v jazyce C++ a pro své vývojáře nabízí zcela unikátní způsob vývoje jejich kódu za pomocí takzvaných **blueprintů** [6], které tvoří základ celého vývoje v prostředí Unreal Enine 4. Více informací o enginu naleznete zde [3].

Tato práce se ve své první části věnuje popisu vytvoření síťové hry v prostředí Unreal Engine 4. Obsahuje detailní postup vytvoření takové hry a je zde také ukázáno a rozebráno několik variant připojení. Další část na tu první navazuje, jelikož se zabývá vytvořením inventáře a systému vyrábění věcí. Obě tyto složky jsou pak implementovány v rámci síťové hry. Dále je v této kapitole popsán postup vytvoření komunikace v síťové hře. Následující kapitola poté shrnuje kapitoly předešlé a obsahuje popis vytvoření hry bludiště, která byla vytvořena za účelem zkoumání chování lidí v bludišti. V poslední kapitole je poté popsáno vytvoření modelu nové chladící budovy nacházející se v jaderné elektrárně Dukovany, který byl použit v rámci projektu **Virtuální prohlídka jaderné elektrárny**.

2 Multiplayer v Unreal Engine 4

V dnešní době obrovského rozmachu počítačových her a simulací se na tyto produkty kladou stále větší požadavky a jeden z těchto požadavků je bezpochyby ten, aby daná hra či simulace fungovala po síti. Spousta uživateků tuto možnost cíleně vyhledává, jelikož například taková hra obsahuje prvky jako soutěživost, spolupráci, možnost si zahrát se svými kamarády a hlavně dokázat si, že jste lepší než váš soupeř, což klasická hra pro jednoho hráče nedokáže nabídnout. Neplatí to ovšem pouze pro hry, ale i pro spoustu pokusů a simulací je nutné, aby se jich mohla zúčastnit větší skupina lidí. UE4 možnost vytvoření takovéto aplikace nabízí a obsahuje spoustu funkcí a metod pro správnou komunikaci mezi klientem a serverem. Veškeré potřebné informace pro tvorbu aplikace pro více hráču naleznete zde [4].

2.1 Základní model

Důležitou znalostí při vytváření síťové hry, je její samotný model, na kterém je daná hra postavena. Multiplayer v UE4 je postaven na modelu klient-server, to znamená, že zde bude jeden server, který bude autoritou (bude provádět veškeré důležité operace) a bude se starat o to, aby všichni klienti, kteří jsou k němu připojení, jsou neustále uvědomováni o dění ve hře a jsou jim neustále posílány veškeré potřebné informace o serveru.

Ve hře pro jednoho hráče je svět reprezentován pomocí jednotlivých aktorů[1]. Ve hře více hráčů to je velice podobné, rozdíl je ale v tom, že jednotliví klienti mají pouze aproximovanou verzi každého aktora, zatímco server si udržuje svou autoritativní verzi. Aktoři jsou v podstatě nejdůležitější prvky celé hry, u kterých se server stará, aby měli o nich klienti nejaktuálnější informace. Pokud je na čase, aby server aktualizoval nějakého konkrétního klienta, tak server nejdříve posbírá veškeré aktory, kteří se nějakým způsobem změnili od poslední aktualizace, a poté klientovi pošle dostatek informací o těchto aktorech, aby měl o nich neaktuálnější přehled.

Pro lepší přehled o modelu klient-server a hlavně o tom, jak k takovému připojení jednotlivých klientů dochází, jsou zde uvedeny základní kroky, které klient a server musí vykonat, aby došlo k úspěšnému připojení klienta na server. Tyto kroky budou detailněji rozebrány v následujících kapitolách.

Prvním krokem je tedy samotné poslání žádosti klienta serveru, že se k němu chce připojit. Server má nyní dvě možnosti, buď žádost povolit nebo zamítnout. Zde samozřejmě hraje roli, jak je server nastavený na přijímání takovýchto zpráv. Dochází zde ke kontrole, zda by nový klient nepřekročil maximální počet povolených hráčů ve hře, nebo jeho odezva je příliš velká a podobně. Pokud tedy server žádost příjme, pošle klientovi aktuální mapu, na kterou se má připojit. Poté server počká, až si tuto mapu klient načte. Jakmile je mapa načtená, server zavolá funkci **AGameModeBase::Prelogin**, která dá šanci třídě **GameMode** zamítnout připojení klienta na server. Pokud se tak ovšem nestane, tak je volána funkce **AGameModeBase::Login**.

[1] Aktor je v Unreal Enginu jakýkoliv objekt, který může být umístěn do scény. Tyto objekty podporují 3D transformace jako jsou rotace, změna měřítka nebo transformace.

Tato funkce má za úkol vytvořit třídu **PlayerController**, která je následně replikována nově připojenému klientovi. Jakmile k tomu dojde, tato třída nahradí dosavadní **PlayerController**, který klientovi sloužil během procesu připojování se na server. V této chvíli je také volána funkce **APlayerController::BeginPlay**. Jedná se o funkci, která jakoby uvede daný **PlayerController** do provozu. Ačkoliv se zdá, že klient již má vše potřebné k tomu, aby mohl plně fungovat, opak je pravdou. V této chvíli se ještě nedoporučuje volat replikované události na daný klientův **PlayerController**, jelikož ještě nebyla volána funkce **AGameModeBase::PostLogin**, jenž připojení nového klienta na server definitivně potvrdí. [5]

2.2 Replikace

Pojmem replikace je v UE4 myšlen již zmíněný proces přenosu informací mezi serverem a klientem. Snadno se to dá pochopit ze situace, kdy se budeme nacházet v síťové hře, kde na nás běží dva útočníci a střílí po nás. Jak je možné, že je oba vidíme a slyšíme jejich střelbu, když celá hra se zpracovává na serveru a ne na našich lokálních počítačích?

Tady právě přichází na řadu replikace. Oba dva protivníky můžeme vidět, protože server vyhodnotil jejich pozici vůči nám jako takovou, že mezi námi není žádný objekt, který by blokoval výhled a tudíž informace o těchto dvou aktorech replikoval na klienta, což jsme v tomto případě my. Jinak řečeno teď máme lokální kopie těchto dvou aktorů. Můžeme vidět, jak na nás běží, protože server replikuje jejich momentální pozici a informaci o jejich tvaru a rychlosti směrem k nám. Mezi jednotlivými aktualizacemi ze strany serveru si poté klient simuluje pohyb oněch dvou protivníků lokálně. Co se týče jejich střelby tak tu jsme schopni slyšet, jelikož server replikuje funkci „ClientHearSound“ směrem k nám. Tato funkce se pak na klientovi volá pokaždé, když server usoudí, že hráč v danou chvíli slyší nějaký zvuk.

Ve shrnutí to tedy znamená, že server jednoduše aktualizuje stav hry a dělá důležitá rozhodnutí. Server také replikuje klientům některé aktory. Dále také replikuje některé proměnné a v neposlední řadě replikuje volání některých funkcí.

Samozřejmě, že kvůli zmenšení nároků na server a hlavně kvůli bezpečnosti úniku důležitých dat, respektive dat potřebných pro chod hry, ne všichni aktoři, proměnné nebo funkce budou replikovány. Představme si například situaci, kdy se nacházíme na jedné straně herní mapy a druhý hráč je na straně druhé a navíc je mezi námi hora nebo jiná překážka. Je jasné, že z naší pozice nemáme šanci daného hráče vidět, a není tedy třeba, aby informace o jeho vzhledu, pozici nebo případně počtu životů nám byly ze serveru replikovány, jelikož pro nás momentálně nejsou důležité. Tímto je možno ušetřit serveru spoustu práce, kdy replikuje pouze ta data, které jsou pro klienty nezbytně nutné, aby se v dané hře orientovali. To samé platí i pro jednotlivé proměnné, ani ty nemusí být všechny replikovány. Jako příklad můžeme uvést proměnnou, pomocí které server určuje chování umělé inteligence, která se ve hře nachází. Ani v tomto případě klient nepotřebuje znát hodnotu této proměnné, jelikož mu to nijak nepomůže a je pro něj zcela nedůležitá. Klient potřebuje pouze proměnné, které jsou ve hře nějakým způsobem zobrazovány, například počet životů nad hlavou hráče nebo proměnné se kterými on sám dále

pracuje. Co se týče funkcí tak u nich platí, že většina těch, které jsou volány na straně serveru není replikována, protože se z velké části jedná o funkce pro chod samotné hry a do těch klient vidět nemusí. Klientu je třeba replikovat pouze ty funkce, které ovlivňují to, co klient vidí nebo slyší, čili funkce, které ho přímo ovlivňují. Ve shrnutí by se tedy dalo říct, že server zpracovává a uchovává velké množství dat a informací, z nichž je pro klienta důležitá pouze malá část, která je následně replikována.

Neexistuje žádný algoritmus nacházející se v Enreal Enginu 4, který by byl schopen sám určit, které aktory, proměnné nebo funkce je třeba replikovat a je to tedy zcela v rukou autora samotného kódu Blueprintu[2], aby o tomto rozhodoval.

2.2.1 Implementace replikace

Na správné implementaci replikace stojí celá síťová hra a je tedy nezbytně nutné si ujasnit určité informace spojené s touto implementací. Jedna z nejdůležitějších částí implementace síťové hry je správné rozvržení dat do tříd, které jsou k tomuto účelu určeny. Pomocí těchto tříd jsme totiž schopni si velice přehledně uspořádat logiku síťové hry a hlavně si rozdělit, která část logiky se bude odehrávat na serveru, a která na klientovi.

Jedná se tedy o následující třídy:

- GameInstance
- GameMode
- GameState
- PlayerController
- PlayerState

První velice důležitá třída je třída **GameInstance**. Tato třída je vytvořena a inicializována ještě před tím než se hráčům načte samotná mapa (level). Jedná se o globálně přístupný Unreal Engine objekt, který může ukládat jakákoliv data, která bychom potřebovali přenášet mezi jednotlivými mapami. Místo abychom museli ukládat data do souboru a poté je znovu načítat pokaždé, když bychom se přemísťovali mezi jednotlivými mapami, tato třída se o to postará za nás. Jako příklad poslouží nějaký předmět nacházející se ve venkovním levelu. Hráč připojený do hry se rozhodne, že vstoupí do budovy. Tato budova bude mít ovšem svůj vnitřní level, který se načte, když do ní hráč vstoupí. Uvnitř tohoto levelu udělá hráč určitou akci, která opět ovlivní předmět nacházející se ve venkovním levelu. Tato informace se ovšem mezi těmito levely pravě pomocí třídy **GameInstance** přenese a tudíž změna provedena hráčem na daném předmětu se promítne ve venkovním levelu, i když se tam hráč právě nenachází. Jedná se pouze o jeden z mnoha způsobů, jakým te tato třída dá využít. [7].

[2]Vizuální skriptovací systém Unreal Enginu [6]

Další třídou v pořadí je třída **GameMode**. Každá hra obsahuje řadu pravidel, podle kterých se řídí a právě tato pravidla tvoří třídu **GameMode**. Na té nejzákladnější úrovni tato pravidla zahrnují například maximální počet hráčů, kteří ve hře mohou být, jakým způsobem se hráči do hry připojují nebo odpojují a další. Důležité je zmínit, že objekty nacházející se v této třídě existují pouze na straně serveru, což je poměrně logické.

Druhá třída, která se stejně jako třída **GameMode** stará o udržování informací o probíhající hře, je třída **GameState**. Pokud se provádějí události založené na herních pravidlech definovaných v předešlé třídě, je třeba informaci o těchto událostech uložit a sdílet s ostatními klienty. To se právě dělá prostřednictvím třídy **GameState**. Například pokud se provede připojení nového hráče a provede se tak funkce ze třídy **GameMode**, tak třída **GameState** může například začít ukládat informace o tom, jak je daný hráč dlouho připojen na serveru nebo kdy se na server připojil. Objekty obsažené v této třídě existují samozřejmě na serveru, ale na rozdíl od třídy předešlé i na všech klientech, aby server mohl používat replikované proměnné a uvědomovat tak všechny klienty o aktuálním stavu hry.

Následující třídou je třída **PlayerState** a už z názvu vypovídá, že uchovává informace o stavu jednotlivého hráče. Stejně jako třída **GameState**, tak i tato existuje jak na serveru, tak na všech připojených klientech. Může se využívat například pro ukládání důležitých informací o každém hráči, které by bylo dobré dál replikovat ostatním, aby o sobě navzájem měli přehled. Jako příklad může být skóre hráče, nebo jeho počet životů.

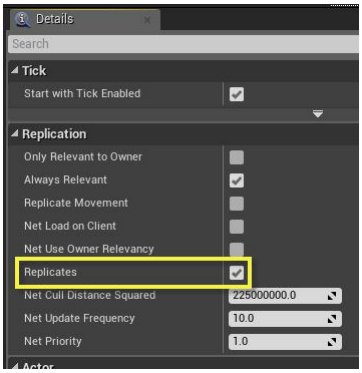
Poslední důležitou třídou je třída **PlayerController**. Tato třída existuje na serveru pro každého hráče, který je do hry připojen. Na straně klientů existují pouze ty **PlayerControllery**, které jsou přímo vlastněny daným klientem. To znamená, že tato třída není nejlepším místem, kde uchovávat důležitá data, která chceme replikovat mezi ostatní hráče, na to je mnohem lepší třída předešlá, čili **PlayerState**. Třída **PlayerController** slouží k řízení akcí a funkcí aktora, ke kterému se váže. V této třídě se tedy většinou nacházejí funkce typu co se stane, když hráč zmáčkne dané tlačítko na klávesnici, nebo co se stane, když daný hráč provede danou činnost a podobně. Dále se tato třída velice využívá při přenosu hráčů mezi větším počtem map. Jelikož hráč, který se nachází například na mapě, kde je jeho úkolem pouze si vybrat novou postavu do hry, tudíž se zde nemůže nijak pohybovat, ani provádět funkce, které by v rámci hry normálně provádět, má přiřazen svůj **PlayerController**, který definuje, jak už bylo zmíněno, co daný hráč může provádět, což je v tomto případě pouze vybrat si postavu. Poté bude přenesen do herní mapy, kde už může provádět veškeré činosti, které hra nabízí a tudíž mu je přidělen nový **PlayerController**, který již tyto funkce obsahuje. Je to mnohem efektivnější a přehlednější způsob předávání funkcionality hráči, nežli povolování a zakazování různých funkcí v různých mapách. Tato změna **PlayerControlleru** se dá samozřejmě využít i jinde, než pouze při změně mapy a to například, když hráč nasedne do auta nebo do jiného dopravního prostředku, tak mu je přidělen **PlayerController** obsahující funkce pro ovládání daného vozidla. Více o těchto třídách naleznete zde [7].

Po pochopení těchto základních tříd, je možné začít se samotnou implementací replikace

pomocí blueprintů v prostředí UE4. Pro odlišnost implementace replikace jednotlivých částí síťové hry, je tato implementace v rámci práce rozdělena na tři hlavní podkapitoly.

Replikace aktora

Replikace aktorů je bezpochyby jedna z nejdůležitějších replikací a tvoří základ síťové hry v prostředí Unreal Engine 4. Pokud je totiž ve scéně umístěn nějaký aktor, například výbušnina, a hráč bude chtít tuto výbušninu odpálit, potom na serveru uvidí celý proces výbuchu i s následným zmizením dané výbušniny ze scény, ovšem na straně žádného z klientů nebude změna viditelná a výbušnina zůstane neporušená na svém místě. Je tomu tak proto, že server neposlal klientovi informace o změně daného aktora a ten tedy nemá žádnou šanci zjistit, co se s výbušninou stalo. Unreal Engine 4 nabízí velice jednoduché řešení jak tento problém vyřešit. Každý aktor, který byl námi vytvořen, má totiž ve svém panelu detailů kolonku **Replicates**, která když se nastaví na **true**, tak server automaticky posílá všem připojeným klientům aktuální informace o daném aktorovi. Jak tento panel vypadá můžete vidět na následujícím obrázku.

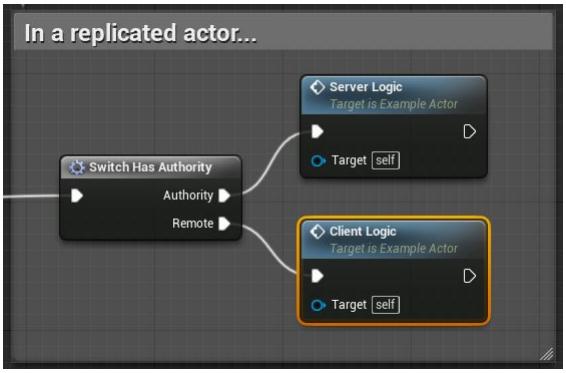


Obrázek 1: Replikace Aktora [8]

Důležité je taky si uvědomit, že aktoři jsou replikováni pouze ze serveru na klienty a nikoliv naopak. Samozřejmě, že klienti mohou serveru posílat data a to prostřednictvím funkcí s nastaveným typem replikace **Run on Server**. K tomu co toto nastavení slouží se dostaneme později.

Každý aktor, který se v dané hře nachází, má také někoho, kdo mu je v dané chvíli autoritou. Co se týče serveru, tak všichni aktoři, kteří se na něm nacházejí a všichni ti, kteří mu byli replikováni, mají jako autoritu právě server. Klienti mají autoritu pouze nad těmi aktory, kteří byli vytvořeni přímo na jejich straně. Tato vlastnost autority se dá využít v případě, že je potřeba, aby daný aktor provedl akci jak na klientovi, tak na serveru, ale v obou případech s jiným výsledkem nebo chováním. Nejlépe to jde vidět na příkladu, kde v našem světě bude umístěný nějaký předmět, který mají hráči za úkol sbírat a pokaždé když tak učiní, dostanou jeden bod a daný předmět zmizí, aby ho nemohl vzít někdo jiný. Dále na severu bude dán požadavek, aby vždy, když hráč tento předmět sebere, se na obrazovku vypíše, kdo daný předmět

sebral a kde se nacházel. Samozřejmě, že tyto informace nebudou zobrazovány na klientech, aby nevěděli pozici svým protihráčů, ale budou zobrazovány pouze na straně serveru, který tyto informace může použít například jako kontrolu, zda daný hráč nepodvádí a má skutečně pouze takový počet bodů, kolik posbíral daných předmětů. Bude tedy potřeba zařídit, aby se předmět choval jinak na serveru než na klientovi. To se dá docílit pomocí funkce, kterou UE4 nabízí a která se nazývá **Switch Has Authority**. Tato funkce rozděluje kód, který se spustí, když někdo sebere daný předmět, na dvě části. První část patří tomu, kdo má nad daným aktorem autoritu (v tomto případě serveru) a druhá část patří tomu, kdo danou autoritu nemá. Na obrázku níže můžete vidět použití této funkce v prostředí UE4.



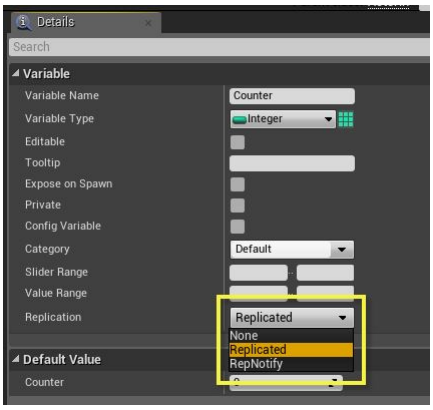
Obrázek 2: Switch Has Authority [8]

Je také dobré vědět, že pokud vytvoříme nového aktora na serveru a ten bude mít vlastnost replikace, potom si každý klient vytvoří na své straně jeho kopii. Ovšem autoritu nad tímto aktorem má stále pouze server. Pokud ovšem vytvoříme aktora na straně klienta, tak i když mu nastavíme, že se má replikovat, tak bude tento aktor existovat pouze na daném klientovi a nikde jinde se jeho kopie nevytvoří. To je způsobeno tím, že replikace probíhá pouze ze serveru na klienta. Nicméně daný klient, na kterém byl aktor vytvořen bude mít nad ním autoritu. To se může hodit u aktorů sloužící pouze pro kosmetické účely hry, například vytvoření animace na straně klienta po dokončení daného úkolu ve hře. Tuto animaci tedy uvidí pouze daný klient a nikdo jiný. pokud ovšem chceme, aby aktor přímo hru ovlivňoval, je dobré ho vytvářet přímo na serveru.

To samé platí o zničení aktorů. Pokud je aktor zničen na serveru, tak všechny jeho kopie nacházející se na všech klientech jsou také zničeny. Klienti mohou zničit aktory, nad kterýma mají autoritu, čili aktory, které sami vytvořili. Pokud se ovšem klient pokusí zničit aktora, který byl vytvořen na serveru a nad kterým nemá autoritu, bude jeho žádost ignorována. Platí zde tedy stejná rada, jako u vytváření nových aktorů. Chceme-li daného aktora zničit, je dobré tak učinit na straně serveru.

Replikace proměnné

Replikace u proměnných probíhá velice podobně, jako tomu bylo u předešlých aktorů. Na začátku je stejný problém. Máme například proměnnou, která představuje počet životů daného hráče a tuto hodnotu budeme chtít zobrazit nad hlavou každého hráče, aby všichni toto číslo viděli. Řekněme tedy, že hráč hostující danou hru dostane zásah. Zmenší se mu tedy počet jeho životů, ale ostatní hráči, kteří se v jeho blízkosti nacházejí, žádnou změnu nezaregistrují. Nebudou tedy vědět, zda ho svým výstřelem zasáhli nebo zda je chyba někde jinde. Abychom byli schopni posílat data ze serveru na klienty, je nutno tuto vlastnost u dané proměnné nastavit. Opět zde platí pravidlo, že replikace probíhá ze serveru na klienta a nikoliv obráceně. Pokud budeme chtít posílat data z klienta na server, tak se tak dá učinit pomocí již zmíněných **Run on Server** funkcí. Je důležité také zmínit, že kdykoliv vytvoříme novou proměnnou, je u ní tato vlastnost natavena tak, aby žádné informace neposílala a je tedy dobré, si při každém vytvoření proměnné uvědomit, zda má být nebo nemá být replikována. Panel detailů takovéto proměnné můžete vidět na následujícím obrázku.



Obrázek 3: Replikace proměnné [8]

Na obrázku 3 lze na první pohled vidět, že možností replikace je tu o jednu více, než tomu bylo u replikace aktorů. První možnost **None** samozřejmě znamená, že nechceme, aby tato proměnná byla replikována, což může být vyžadováno u proměnných, které nějakým způsobem server potřebuje pro chod hry a není žádoucí, aby o nich klient věděl. Druhá možnost **Replicated** už znamená, že proměnná bude replikována na všechny klienty připojené do hry, čili se jedná o stejné nastavení replikace jako u předešlých aktorů. Poslední možností je **RepNotify**. Pokud je zvolena tato možnost, UE4 sám vytvoří automaticky funkci s názvem **OnRep_NázevProměnné**. Tato funkce je poté volána pokaždé, pokud dojde ke změně této proměnné a ta je poté replikována na klienty. Jako příklad je možno uvést hru, kde hráči mají za úkol sbírat předměty po mapě a pokaždé, když tak učiní, se jim jednak zvětší skóre a jednak se naplní nádoba s vodou, kterou každý z hráčů musí naplnit, aby vyhrál. Dalo by se to samozřejmě řešit přímo v kódu daného předmětu, který pokaždé, když by byl sebrán, by zavolal funkci, která by následně naplnila danou

nádobu s vodou. Ovšem v případě, že by předmětů bylo více nebo by bylo více možností, jak získat body a naplnit tak nádobu, je rozhodně lepší a časově výhodnější, aby funkce naplnění nádoby byla volána vždy, když se změní proměnná skóre. A přesně k tomu by sloužilo toto nastavení pro replikaci proměnných, kde by se kód řídící naplnění dané nádoby prováděl uvnitř funkce **OnRep_NázevProměnné**.

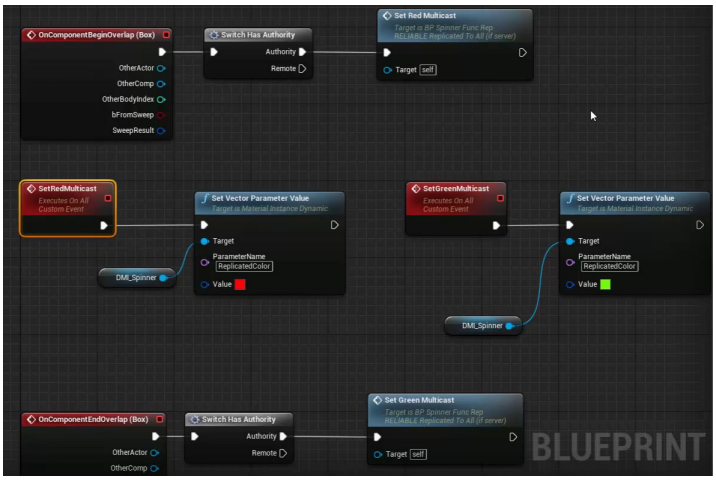
Replikace událostí

Replikace funkcí neboli událostí, jak se označují v prostředí UE4, je složitější a obsáhlejší než replikace předešlé. Je zde totiž více způsobů replikace, než tomu bylo v předešlých kapitolách. Opět bude nastíněn základní problém. Řekněme, že bude potřeba vytvořit funkci provádějící určitou akci. V prostředí UE4 je již spousta funkcí naimplementovaných a jsou k dispozici prostřednictvím blueprintů. Ovšem i tyto samotné funkce mohou být volány v rámci námi vytvořených funkcí v závislosti na tom, jestli mají být volány na serveru nebo klientovi, jelikož u funkcí již naimplementovaných se tato replikace nastavit nedá. Od tohoto nastavení se poté odvíjí, jak se na zmíněných dvou stranách budou chovat jednotlivé části kódu v nich obsažených. Na dalším obrázku lze vidět, jak vypadá panel detailů, takto vytvořené události.

Obrázek 4: Replikace událostí [8]

První možnost je možnost, která je nastavena po vytvoření každé události. Jedná se o možnost **Not Replicated**. Znamená to tedy, že daná událost nebude nijak replikována. Opět se jedná o nastavení, které můžeme použít v případě, že se jedná o událost nijak neovlivňující chod hry, jejíž výsledek není třeba posílat ostatním hráčům. Druhá možnost se nazývá **Multicast**. Pokud je událost s tímto nastavením vyvolána na serveru, je poté replikována na všechny klienty, kteří jsou připojení do hry, bez ohledu ne to, kdo je vlastníkem objektu, který událost ovlivňuje. Pokud je ovšem tato událost volána na straně klienta, bude se chovat, jakoby nebyla replikována, a bude provedena pouze na klientovi, který ji vyvolal. Pokud tedy chceme, aby všichni klienti věděli a výsledku a průběhu dané události, je dobré, aby události typu **Multicast**, byly vždy volány na serveru. Jednoduchým příkladem může být obyčejná kostka nacházející se ve hře. Pokaždé, když hráč vstoupí do prostoru kolem kostky bude volána funkce pro změnu materiálu na červenou a pokaždé, když prostor opustí bude volána funkce pro změnu materiálu zpět na zelenou. Budeme samozřejmě chtít, aby všichni hráči připojení do hry viděli danou změnu. Právě toto umožňuje událost typu **Multicast**. Je o všem nutné dodržet již zmíněné pravidlo,

které říká, že funkce musí být volána na serveru, jinak se provede pouze na klientovi, který ji vyvolá a změna nebude replikována, což způsobí, že samotný efekt vyvolaný touto událostí, bude viditelný pouze pro toho klienta, na kterém byla událost volána. Obrázek níže ukazuje, jak tento kód může vypadat v blueprintech.



Obrázek 5: Příklad události typu Multicast

Na obrázku 5 je možné vidět, že po zavolání funkce **OnComponentBeginOverlap**, což je funkce, která se spustí, když hráč vkročí do prostoru s názvem **Box**, nacházející se kolem kostky, dochází k zavolání již známé funkce **Switch Has Authority**, která zajistí, že následující část kódu se provede na straně osoby, která má nad aktorem kostky autoritu, což je server. Poté následuje zavolání samotné funkce měnící materiál. To samé se provádí i dole na obrázku v případě, že hráč prostor opouští. Uprostřed obrázku se poté nachází samotná implementace funkce měnící materiál, která má nastavení replikace nastavené právě na **Multicast**. Funkce s tímto nastavením se velice snadno používají ovšem mají jednu nevýhodu a tou je zatížení serveru. Jak už bylo zmíněno, tato funkce se replikuje pokaždé, když je zavolána na serveru na všechny klienty připojené do hry, což v případě, že se jedná o událost jako je například střelba, která se provádí několikrát za sekundu na velkém množství míst ve hře, je velice neefektivní. UE4 má proti velkému počtu volání těchto událostí naimplementovanou obranu, která ignoruje volání těchto událostí, pokud překročí určitý limit volání v rámci malého časového intervalu.

Druhým typem replikace událostí je **Run on Server**. Tento typ událostí byl již zmíněn ve spojitosti s replikací aktorů a proměnných, jako způsob posílání dat opačným směrem, než je směr replikace, čili směrem od klienta na server. Pokud je tato událost vyvolána na serveru, tak poběží pouze na serveru, ovšem pokud je vyvolána na klientovi, tak bude replikována a spuštěna na serveru. Příkladem zde může být situace, kdy se na serveru nachází vlajka, kterou mají hráči za úkol zajmout. Pokud k ní první hráč přiběhne a bude ji chtít obsadit, on sám nemůže vyvolat funkci pro změnu její barvy, nebo chování, protože není jejím vlastníkem, jelikož vlajka byla vytvořena serverem a on má nad ní autoritu. Místo toho vyvolá na straně klienta událost typu **Run on Server**, která jakoby řekne serveru něco ve smyslu „Nacházím se na pozici u vlajky,

prosím zavolej funkci obsazení“. Tato událost se poté replikuje na server a je na něm zavolána, což už barvu vlajky změní, jelikož server je jejím vlastníkem. Událost tohoto typu může být tedy použita pouze na aktory, kteří jsou vlastněni nějakým **Player Controllerem** nebo jsou jim sami.

Posledním typem událostí jsou události typu **Run on Owning Client**. Jedná se v podstatě o opak předešlého typu událostí. V tomto případě jsou události vyvolané na serveru replikovány a spouštěny na klientovi, který vlastní daného aktora. Jelikož server také může vlastnit aktory, tak i tento typ událostí může v případě, že se jedná o aktora vlastněného serverem, být zavolán právě na serveru, i když název říká něco jiného. Pokud je ovšem tato událost vyvolána na klientovi, je brána jako nereplikovaná a je zavolána pouze na klientovi, který ji vyvolal.

Následující dva obrázky ukazují, jak různé typy replikace ovlivňují, kde se bude daná událost spouštět, v závislosti na místu vyvolání. První obrázek ukazuje události vyvolané na serveru a druhý události vyvolané na straně klienta.

	Not replicated	Multicast	Run on Server	Run on Owning Client
Client-owned target	Server	Server and all clients	Server	Target's owning client
Server-owned target	Server	Server and all clients	Server	Server
Unowned target	Server	Server and all clients	Server	Server

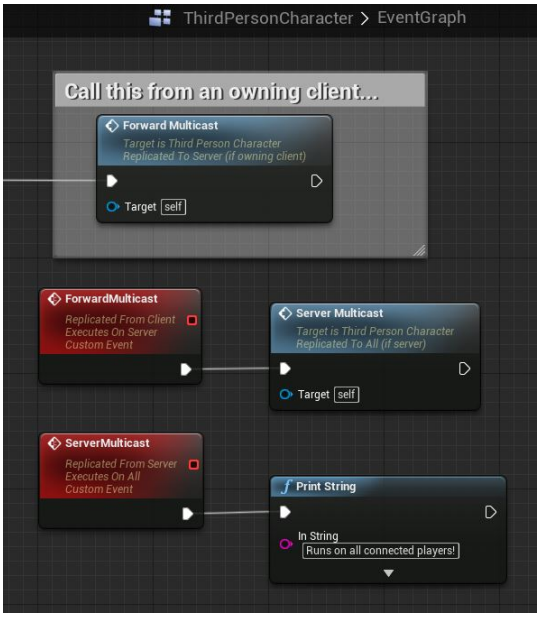
Obrázek 6: Závislost místa běhu události na typu replikace. Vyvoláno na serveru [8]

	Not replicated	Multicast	Run on Server	Run on Owning Client
Target owned by invoking client	Invoking client	Invoking client	Server	Invoking client
Target owned by a different client	Invoking client	Invoking client	Dropped	Invoking client
Server-owned target	Invoking client	Invoking client	Dropped	Invoking client
Unowned target	Invoking client	Invoking client	Dropped	Invoking client

Obrázek 7: Závislost místa běhu události na typu replikace. Vyvoláno na klientovi [8]

Z obrázku 7 je možné vyčíst vyčíst, že všechny události vyvolané na klientovi, které nejsou nastaveny na **Run on Server**, se chovají, jako nereplikované. Proto pokud je to možné, všechny události vyvolané na klientovi by toto nastavení mít měly, pokud chceme, aby jejich výsledek byl znám i ostatním hráčům připojeným do hry. Posílání replikované události z klienta na server je totiž jediná možnost, jak komunikovat mezi klientem a serverem v tomto směru. Také je dobré znovu připomenou že události typu **Multicast** mohou být volány pouze ze serveru. Jelikož základní model pro síťovou hru, který nabízí UE4 je model klient-server, tak klient není přímo propojený s ostatními klienty, ale pouze se serverem. Proto klient není schopen poslat **Multicast** události přímo ostatním klientům, ale musí komunikovat se serverem, který už toho schopen je. Toto pravidlo se ovšem dá obejít spojením dvou událostí do jedné. Nejprve je vytvořena událost typu **Multicast**, která bude provádět logiku celé události, například tisknut text na obrazovku. Tuto událost budeme chtít vyvolat klientem. Ovšem již víme, že pokud chceme komunikovat směrem od klienta k serveru, musíme to provádět pomocí události **Run on Server**. Čili si tuto událost vytvoříme a v rámci ní potom zavoláme naší předešlou událost, která již je **Multicast**.

To způsobí, že i když voláme událost typu **Multicast**, pošleme o ní informaci na server a ten už se postará o to, aby o jejím výsledku věděli všichni připojení klienti. Následující obrázek ukazuje, jak by to mohlo vypadat.



Obrázek 8: Volání Multicast události ze strany klienta [8]

Jako poslední věc, kterou je třeba zmínit v souvislosti s replikací, je situace, kdy se hráč připojí již do probíhající hry. Pokud se totiž hráč připojí, veškeré replikované události, které se provedly před tím, než se hráč dostal do hry, nebudou vyvolány na nově příchozím hráči. Nejlepší způsob jak zajistit aktuální informace pro nově příchozí hráče, je udržovat důležitá herní data v replikovaných proměnných. Tento způsob je velice používaný a spočívá v tom, že pokud nějaký hráč provede určitou akci a informuje o ní server prostřednictvím **Run on Server** události, tak v rámci této události server aktualizuje některé ze zmíněných replikovaných proměnných v závislosti na provedené akci. Další hráč, který nevyvolal tuto akci, potom vidí výsledky této akce právě prostřednictvím daných proměnných. Navíc hráč, který se poté připojí do hry, může vidět aktuální stav hry, jelikož mu server posílá aktuální hodnoty replikovaných proměnných.

Reliability

Reliability neboli česky spolehlivost, je jedním z dalších nastavení, které musíme zvážit při nastavování replikace. Tato vlastnost se dá nastavit u všech replikovaných událostí a může mít pouze dvě hodnoty a to **true** nebo **false**. U proměnných je automaticky nastavená na **true**, bez možnosti změny. Co se ovšem týče událostí, tak pokud nastavíme, že chceme, aby daná událost byla „spolehlivá“, garantujeme tím, že tyto události vždy dosáhnou svého cíle, za jakýchkoliv podmínek. Samozřejmě toto má za následek větší zátěž na server, který se i v případě vysokého náporu na síť bude snažit, aby se daná událost správně provedla. Další negativní vliv, který to

může přinést je například latence. Je tedy dobré zvážit, které události budou takto nastaveny a které ne. Jako příkladem mohou být události, které jsou volány v každém snímku probíhající hry, jelikož může dojít k přetížení a klient, jenž je s těmito událostí spojen, bude odpojen od hry. Na druhou stranu události, které se starají například o zobrazování životů hráče nebo o jiná důležitá data, by toto nastavení mít rozhodně měly.

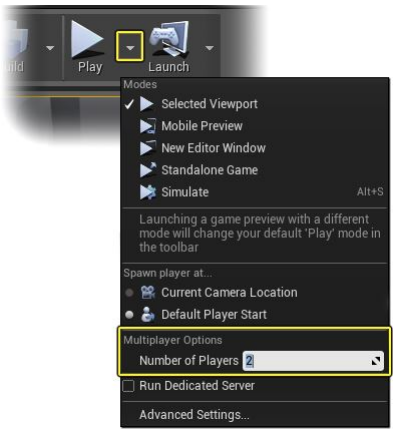
Opakem těchto událostí jsou události „nespolehlivé“. V tomto případě není zaručeno, že se informace dostane na místo určení, například z důvodu ztráty paketů v síti, nebo v případě, že engine určí, že je zde spousta jiných informací s daleko větší prioritou, které je třeba poslat. Nejjednodušším příkladem můžou být jednotlivé části efektů, například když hráč postoupí na novou úroveň. Pokud se některá z těchto částí nezobrazí úplně celá, tak to na hru nebude mít žádný vliv a je veliká šance, že si toho ani žádný z hráčů nevšimne.

2.3 Vytvoření síťové hry

V této podkapitole bude postupně popsáno několik druhů vytvoření základní logiky hostování a připojování se na server a celkový postup vytvoření základů pro hru bludiště, jenž byla základem této diplomové práce.

2.3.1 Testování síťové hry v prostředí Unreal Engine 4

Unrear Engine 4 nabízí velice jednoduchou možnost, jak si hru pro více hráčů otestovat. Není k tomu třeba žádná implementace, ani není nutné mít hotovou logiku připojování se na server. Engine nám totiž umožňuje spustit editor v režimu více hráčů a my si tak můžeme otestovat námi naimplementované události, zda u nich dochází ke správné replikaci informací a zda vše funguje tak, jak má. Na obrázku níže lze vidět, jak jednoduše nastavit, aby hra běžela pro námi zvolený počet hráčů.



Obrázek 9: Testování hry po síti [9]

Pokud hru spustíme v režimu, jaký je zobrazen na obrázku 9, tak se základní okno editoru použije, jako okno pro server a k němu jsou pak vytvořená další okna pro každého hráče připoje-

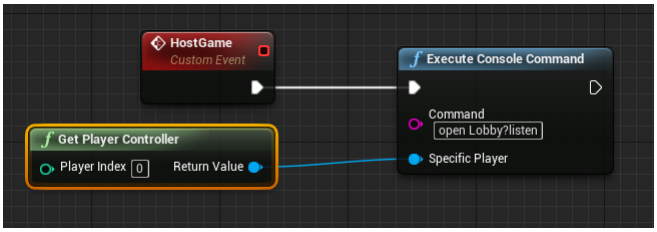
ného do hry. Je zde samozřejmě celá řada nastavení, od velikosti okna, až po pozici jednotlivých oken. Co je ovšem mnohem důležitější, je možnost pokročilých nastavení pro hru více hráčů, konkrétně jestli chceme náš server spustit v režimu **Listen Server** nebo **Dedicated Server**. Jedná se o dva základní typy hostování síťové hry. První z nich, tedy **Listen Server** znamená, že stroj, který má autoritu a na kterém běží server, má také své vlastní okno a může hrát danou hru, během jejího hostování. Naproti tomu **Dedicated Server** hru pouze hostuje a tudíž se sám hry nezúčastní. Tato možnost je samozřejmě méně náročná pro server, jelikož nemusí na své straně provádět žádné vizualizace dané scény. Více informací o testování síťové hry lze najít zde [9].

2.3.2 Vytvoření základní logiky připojení

Jak už bylo zmíněno, tak pokud je potřeba přenášet data mezi různým počtem map, tak k tomu slouží třída **GameInstance**. Právě v této třídě by měla být naimplementována celá logika hostování a připojení se do hry, aby bylo možné tyto funkce volat z jakéhokoliv místa v rámci blueprintů. Konkrétně v mém případě v rámci **widgetů**footnoteNástroj Unreal Enginu 4 pro vytváření uživatelských rozhraní. [10] starajících se o zobrazení uživatelského rozhraní pro hlavní menu celé hry.

Hostování pomocí konzolových příkazů

Existuje několik způsobů jak zahájit hostování síťové hry. Ten nejjednodušší a nejzákladnější způsob spočívá pouze v zavolání jednoduchého konzolového příkazu. Unreal Engine 4 má v sobě k dispozici naimplementovanou funkci, která uživatelům umožňuje pracovat přímo s konzolovými příkazy v rámci jejich kódu, respektive blueprintu. Tato funkce se nazývá **Execute Console Command** a očekává dva vstupy. Prvním vstupem je samotný konzolový příkaz, který se má provést a druhým vstupem je **Specific Player**. Tento vstup definuje, který hráč daný příkaz provádí a očekává tedy třídu **PlayerController** daného hráče. Na obrázku níže je možné vidět, jak takové základní hostování hry vypadá.

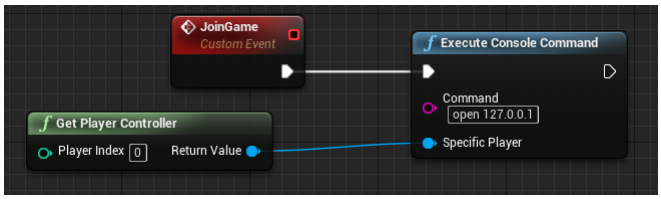


Obrázek 10: Funkce Execute Console Command pro hostování

Na obrázku 10 je vidět, že funkce používá pro jeden ze svých vstupů funkci **Get Player Controller**. Tato funkce vrací třídu **Player Controller** aktuálního hráče, čili hráče, který například stiskl tlačítko vyvolávající zobrazenou událost. Na druhém vstupu si můžeme všimnout

řetězce „*open Lobby?listen*“. První příkaz *open* slouží k načtení a následnému zobrazení mapy, jejíž název následuje hned za tímto příkazem. V našem případě se mapa jmenuje *Lobby*. Následuje příkaz *?listen*, který říká serveru, aby „naslouchal“, zda se k němu nějaký hráč nechce připojit. Jednoduše tento příkaz načte zvolenou mapu v takzvaném listen módu. Je také důležité vědět, že implicitně bude server poslouchat na portu 7777, je tedy dobré tento port na svém routeru povolit nebo změnit na port jiný.

Nyní je na řadě se podívat na to, jak se na server vytvořený tímto způsobem připojit. Postup je naprosto stejný jako u hostování hry, čili pomocí funkce **Execute Console Command** s tím rozdílem, že se nám změní jeden ze vstupů a to konkrétně ten, který zpracovává vstupní konzolový příkaz. Opět zde uvádím obrázek přímo z prostředí Unreal Engine 4 na kterém je takovéto připojení do hry znázorněno.



Obrázek 11: Funkce Execute Console Command pro připojení

Zde na obrázku 11 je možné vidět, že příkaz obsahuje pouze „*open 127.0.0.1*“, kde příkaz *open* opět značí načtení mapy, ovšem místo jejího jména je tento příkaz následován IP adresou daného serveru, na který se chceme připojit. Pokud na straně serveru nedošlo ke změně portu, který je, jak již bylo zmíněno, implicitně nastaven na 7777, tak ani zde se zadávat nemusí, jelikož i zde je implicitní nastavení portu stejné. IP adresa je zde nastavena právě na *127.0.0.1*, protože se jedná o lokální adresu našeho počítače, čili pokud se pokusíme na tuto adresu připojit, tak se připojíme na vlastní počítač. Tato adresa je zde uvedena tedy právě pro testování síťové hry pro více hráčů na jednom počítači.

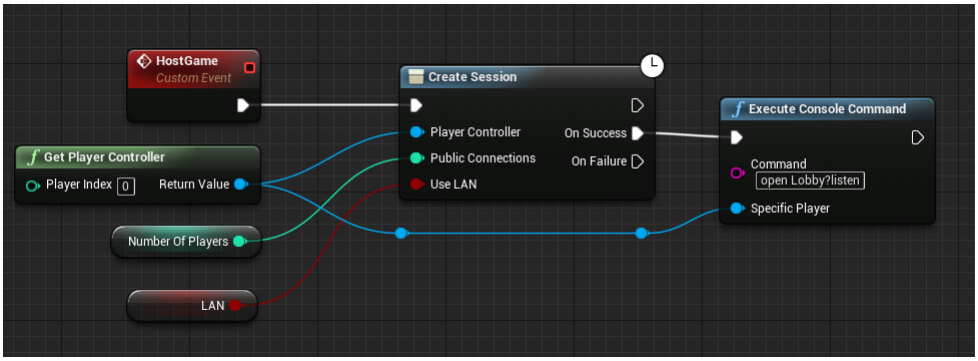
Ačkoliv se může zdát, že tento způsob hostování hry je velice efektivní, jelikož nevyžaduje nijak složitou implementaci, opak je pravdou. Není sice obtížné vytvořit takovýto jednoduchý server, ovšem v případě, že si tuto možnost zvolíme, musíme počítat s tím, že zde nebude moc možností ohledně podrobnějšího nastavení hostování a také zde není snadné přidávat různé vlastnosti, jako například jméno serveru, přímo na daný server. Co přesně se tím myslí si ukážeme hned při popisu druhého druhu hostování, který už tyto vlastnosti umožňuje nastavit.

Použití Online Session Nodes

Druhá možnost hostování už je více propracovaná a nabízí celou řadu funkcí, které se starají o správný průběh připojení hráčů do hry a také o správu chybových hlášek, pokud dojde k nějaké chybě. Skupina těchto funkcí se nazývá **Online Session Nodes** [11]. Jedná se tedy o skupinu „uzlů“, které se dají přímo použít v rámci blueprintů. Jsou zodpovědné za hostování,

hledání aktivních serverů, připojování a nebo odpojování hráče od serveru. Všechny tyto funkce jsou latentní, to znamená, že provádějí svou operaci na pozadí, obvykle je tomu tak, protože tyto operace musejí komunikovat skrze síť a končí poté v určitém okamžiku v budoucnu. To, že je funkce latentní lze v prostředí UE4 zjistit pomocí symbolu hodin nacházející se v pravém horním rohu každé takové funkce.

Stěžejní funkcí této skupiny je funkce **Create Session**. Na dalším obrázku je možné vidět, jak vytvoření hostování pomocí této funkce vypadá.

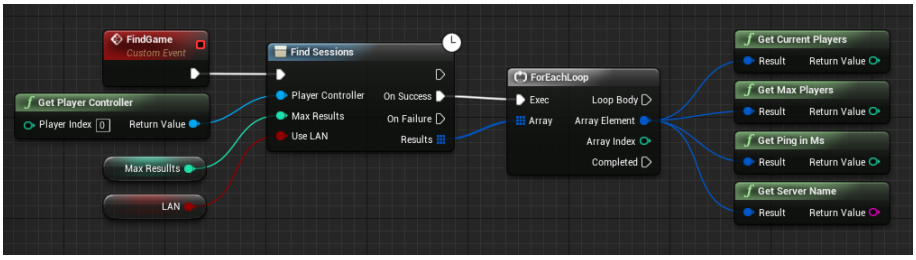


Obrázek 12: Funkce Create Session

Tato funkce nám tedy vytvoří takzvanou **session**, jak už z jejího názvu vypovídá, a tato **session** se poté bude vázat na mapu, jejíž jméno se nachází v následující funkci, o které již byla zmínka v předchozí kapitole. Respektive pokud se hráč bude chtít připojit do hry, nebude již muset zadávat informace jako je IP adresa, nebo dokonce název mapy, jelikož již nehledá, respektive se již nesnaží připojit na tuto mapu, nýbrž na danou **session**, která se již postará, aby se mu načetla správná mapa se správnými informacemi, které obsahuje. Jak je vidět z obrázku 12, tak tato funkce si vyžaduje 3 vstupy. První z nich je již známý. Jedná se opět o referenci na hráče, jenž danou schůzi vytváří, pomocí třídy **Player Controller**. Druhým vstupem je maximální počet hráčů ,který se do hry může připojit a posledním vstupem je proměnná typu true/false, která udává, zda se bude jednat o hostování po internetu nebo po LANu. Už zde jde vidět značná výhoda oproti předešlé metodě hostování.

Druhá výhoda se ukáže hned vzápětí, při popisu následující funkce **Find Session**. Z názvu je jasné, že tato funkce slouží k vyhledávání vytvořených **sessions**. Zde nastává asi největší výhoda této skupiny funkcí, jelikož může nastat situace, kdy budeme mít více vytvořených **sessions** a budeme chtít vypsat jejich přehled pomocí jednoduchého seznamu, abychom si následně mohli vybrat, ke které se připojíme. Přesně tuto možnost funkce nabízí, jelikož vrací pole všech nalezených **sessions**. Předešlá metoda toto neumožňovala, jelikož zadáním IP adresy končí jakákoliv možnost výběru. Další výhodou je, že každá **session** sebou také nese informace o aktuálním počtu hráčů k ní připojených, maximální počet hráčů, odezvu v milisekundách a jméno serveru, což je většinou nahrazené názvem počítače, na kterém je hra vytvořena, následovaným číslem odlišujícím od sebe větší počet **sessions** vytvořených na stejném počítači. Tyto informace jsou

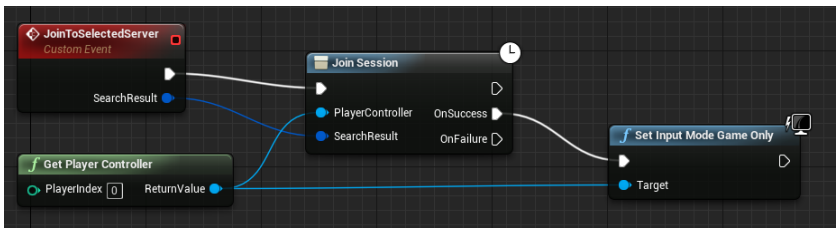
velice důležité, jelikož se dají využít pro nastavení omezení pro hledání **sessions**, které mají například větší odezvu než námi zvolené číslo. Pro představu, jak použití této funkce vypadá v prostředí UE4, slouží následující obrázek.



Obrázek 13: Funkce Find Session

Funkce opět požaduje na svém vstupu 3 parametry. Jedná se opět o referenci na hráče, který hledání prování, čili o jeho třídu **Player Controller**, dále pak je zde proměnná typu true/false, jenž uvádí, zda se hledání provádí po internetu nebo po LANu. A nakonec od funkce předešlé se liší parametrem prostředním, který udává, kolik **sessions** má tato funkce vyhledávat, respektive po dosažení zadaného počtu, už nebude další hledání probíhat. Jak už bylo zmíněno, tak výstupem této funkce je pole nalezených **sessions**, což lze vidět na obrázku 13, kde je tento výstup označen pojmem **Results**. Je zde také vidět jakým způsobem je možno z nalezené **session** vyčíst ony zmíněné informace. Jednoduchým **for each** cyklem je možno projet jednotlivé výsledky a pro každý z nich si vytáhnout potřebnou informaci.

Další funkce se nazývá **Join Session**. Zde je myslím naprosto jasné, že daná funkce slouží k samotnému připojení hráčů k vybrané **session**. Oproti první metodě hostování, která byla popsána výše, zde hráč nemusí volat žádný konzolový příkaz, nýbrž mu stačí pouze zavolat zmíněnou funkci, která si ze zvolené **session** sama IP adresu obstará a připojí se k ní. Funkce také v sobě již obsahuje příkaz *open*, čili jakmile se hráč připojí do zvolené hry, je mu automaticky načtena příslušná mapa. Opět následuje obrázek zobrazující tuto funkci.

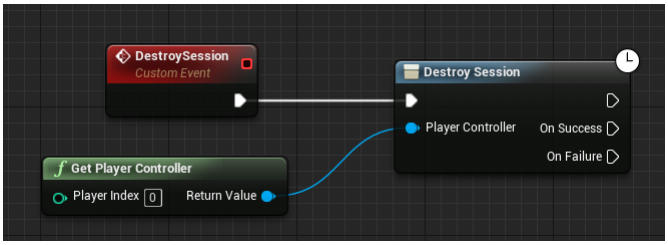


Obrázek 14: Funkce Join Session

Na obrázku 14 lze vidět stále se opakující vstup u tohoto druhu funkcí a tím je **Player Controller**. Stejně jako u funkcí předešlých, i zde je to pochopitelné, jelikož je třeba samozřejmě znát referenci na hráče, který se pokouší připojit do hry. Druhý vstup se nazývá **SearchResult** a je tím myšlena jedna z pole **sessions**, které našla funkce **Find Session**. Respektive se zde zadává, ke které konkrétní **session** se hráč chce připojit připojit. Před každou funkcí jsem si také

vytvořil vlastní událost, která se v tomto případě nazývá **JoinSelectedServer**. Je to z toho důvodu, že pokud bych potřeboval zavolat funkci **Join Session**, stačí mi zavolat tuto mnou vytvořenou událost, která bude jako vstup brát právě zvolenou **session** a tu pak předá funkci **Join Session**. Za touto funkcí již následuje pouze kosmetická záležitost a sice nastavení hráčova „focusu“[3] na samotnou hru, respektive na okno, ve kterém se hra nachází, jelikož po provedení funkce **Join Session** bychom museli tento focus získat kliknutím levého tlačítka na myši, což je velice nepohodlné a nepraktické.

Poslední funkcí z této skupiny je funkce **Destroy Session**. Tato funkce nejenom že shodí danou **session**, pokud je zavolána ze strany serveru, ale je také potřeba zavolat na straně klienta, pokud se samotný klient chce od hry odpojit. Samozřejmě pokud je tato funkce zavolána hostitelem, tak to odpojí nejenom jeho, ale také všechny připojené hráče, jelikož se touto funkcí **session** zcela zruší. Oproti funkcím předešlým je volání této funkce jednoduché a nepotřebuje žádné vstupy kromě **Player Controlleru** hráče, který se chce odpojit od hry. Toto se dá využít v případě, kdy se některý z hráčů nechová podle pravidel hostitele a ten se ho rozhodne ze hry vyhodit. Může tedy zavolat funkci **Destry Session**, kde za již zmíněnou referenci na hráče, dosadí **Player Controller** toho hráče, kterého chce ze hry vyhodit. Funkce tak bude volána pouze na daném klientovi a hra poběží normálně dál. Jak je vidět na následujícím obrázku, volání této funkce není nijak složité.



Obrázek 15: Funkce Destroy Session

Spravování chybových hlášek

Při práci s operacemi pracujícími po síti můžou samozřejmě nastat různé problémy jako například pokus klienta o připojení těsně poté co se host odpojil ze hry, ale seznam serveru se mezitím nestačil aktualizovat nebo můžou nastat problémy s internetovým připojením a nebo spousta dalších problémů s tím spojených. Pokud tvůrce hry chce, aby hra fungovala bez problému a v případě problému hráče dostatečně informovala, můžeme využít funkcí, které UE4 nabízí.

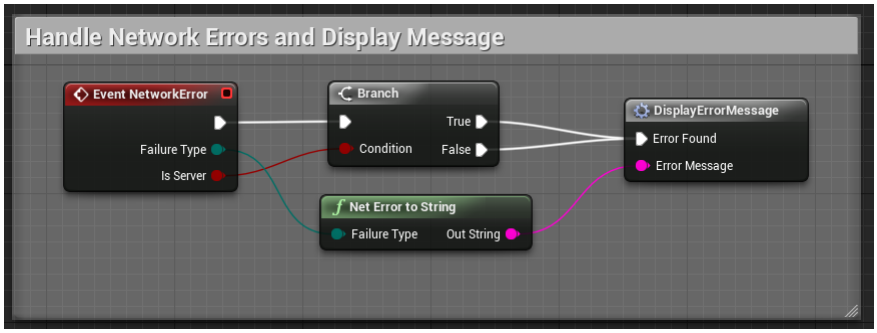
Jako první je možné při hraní síťové hry narazit na chyby, které jsou způsobeny přímo voláním funkcí ze skupiny **Online Session Nodes**. Může se například stát že se **session** nepodaří

[3] V prostředí počítačových her je tímto pojmem myšleno, na kterou obrazovku/okno/část uživatelského rozhraní se hráč soustředí, respektive na které z těchto částí se provede operace pokud k ní dá hráč podnět.

vytvořit, nebo že se již k vytvořené **session** nelze připojit. Přesně z tohoto důvodu mají tyto funkce výstupní piny **OnSuccess** a **OnFailure**, které se provedou pokud se funkce provede správně nebo nastane nějaká chyba. Přímo zde je tedy možno si například vypsat text, který by na danou chybu upozorňoval.

Ovšem ne všechny chyby vznikají pouze na začátku vytvoření **session** nebo na začátku připojení. Většina chyb vzniká přímo během hraní a k tomu toto jednoduché ošetření nebude stačit. K tomuto účelu slouží skupina funkcí **Error Handling Nodes**[11]. Tuto skupinu funkcí je možno volat přímo ze třídy **Game Instance**, což zajišťuje možnost jejich volání z kterékoliv mapy.

První z těchto funkcí se nazývá **Event Network Error**. Tato funkce umožňuje hráči dostat podrobnou informaci o chybě způsobené prováděním operací po síti. Obsahuje dva výstupy. Prvním výstupem je proměnná typu true/false, která se ptá, zda je zpráva o chybě zobrazovaná serveru nebo klientovi. To umožňuje vypisovat různé informace pro obě strany. Pouhý text, že je něco špatně na straně klienta a například detailní výpis chyby na straně serveru. V tomto případě bude za touto proměnnou následovat funkce **Branch**, která rozděluje logiku celé události na dvě části, které se volají v závislosti na hodnotě vstupní proměnné typu true/false. Druhým výstupem je potom samotný **Failure Type**, který vrací druh chyby, která nastala. Tuto proměnnou lze převést do textové podoby a poté ji například vypsat hráči na obrazovku. Typický příklad použití této funkce lze vidět na následujícím obrázku.

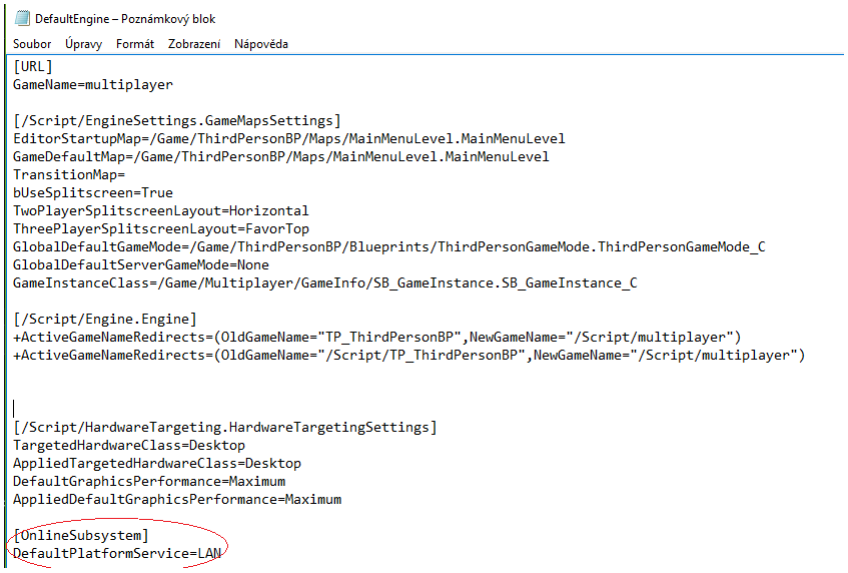


Obrázek 16: Event Network Error

Druhým zástupcem funkcí tohoto typu je funkce **Event Travel Error**. Tato funkce je méně častá, jelikož většina chyb je podchycena již zmíněnou funkcí **Event Network Error**, nicméně se používá v souvislosti s přenosem hráčů mezi jednotlivými mapami. Pokud nastane při tomto přenosu chyba, tato funkce ji podchytí a stejně jako její předchůdce, potom může hráči vypsat danou chybu na obrazovku. Funkce je volána naprosto stejně jako funkce předchozí, s tím rozdílem, že neobsahuje výstupní parametr **isServer**.

Na konec je třeba zmínit, že aby všechny výše zmíněné funkce ze skupiny **Online Session Nodes** správně fungovaly, musí být nastavený, respektive přidán takzvaný **Online Subsystem**. Pro tuto práci není důležité více rozebírat jeho možnosti a způsoby nastavení. Více informací o **Online Subsystému** naleznete zde [12]. Stačí pouze říct, že se jedná o vložení různých

uživatelského rozhraní do naší hry. Jako příklad může být rozhraní Steam nebo Xbox. Pro účely síťové hry ovšem stačí pokud v souboru **DefaultEngine** doplníme text, jak je ukázáno na následujícím obrázku.



Obrázek 17: Online Subsystem

Na obrázku 17 lze vidět, že jako **DefatultPlatformService** byl zvolen **LAN**, což umožňuje fungování výše zmíněných funkcí právě po LANu. V případě, že by uživatel chtěl, aby se daná hra dala hrát například po Steamu, musel by zde místo LANu zadat Steam. Poté by jeho hra byla viditelná pro všechny uživatele používající tuto službu a také jejich připojení skrze tuto službu by nebyl žádný problém.

2.3.3 Přenos hráčů mezi mapami v síťové hře

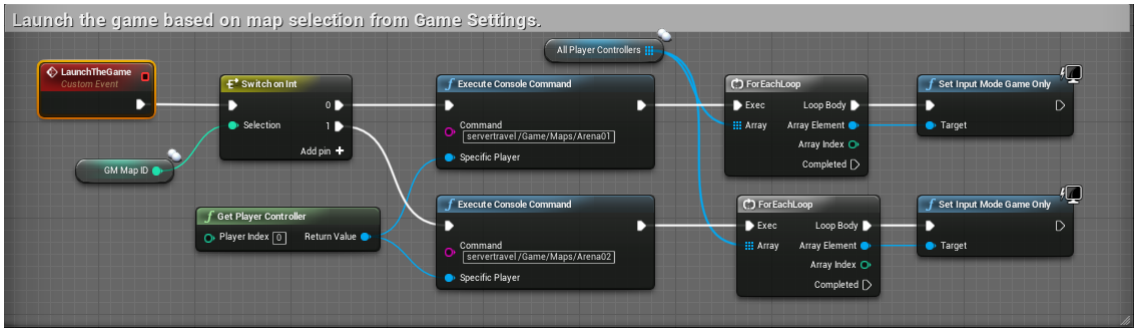
Způsobů jak lze tento přenos provést je několik, ovšem všechny spadají do dvou kategorií a sice **Seamless** a **Non-seamless**. Hlavním rozdílem těchto dvou kategorií je, že **non-seamless** přenos vždy odpojí daného klienta od serveru a poté znovu připojí, již do nové mapy. Z toho důvodu je velice doporučováno používat co možná nejvíce **Seamless** přenos, jelikož se tím výrazně sníží nároky na server. Navíc se dá tímto způsobem také vyhnout celé řadě možných chyb, které by mohly nastat při opakovaném připojování klienta na server[13].

Nyní třeba si vybrat jednu z funkcí, které provedou samotné přenesení hráče. UE4 nabízí celkem 3 funkce, které toto umožňují. První z nich se nazývá **UEngine::Browse**. Tato funkce se bude pokaždé chovat jako **Non-seamless**, jelikož se jedná o něco jako celkový reset, kdy klienti jsou odpojeni od serveru a následně znovu připojeni na novou mapu. Dá se využít, pokud je potřeba přenést pouze jednoho hráče.

Druhá funkce se nazývá **UWorld::ServerTravel**. Už z názvu vypovídá, že tato funkce může být volána pouze serverem. Její zavolání způsobí přenesení serveru na jinou mapu, kde všichni

klienti budou server po takovémto přenesení následovat. Je to využíváno v podstatě ve všech hrách více hráčů, kde se nejedná o přenesení pouze jednoho hráče, ale kdy se mění celá mapa hry. Server nejdříve zavolá tuto funkci na své straně a poté pro každého hráče, připojeného do hry zavolá funkci **APlayerController::ClientTravel**, která způsobí jeho přenesení.

Poslední funkcí je tedy již zmíněná funkce **APlayerController::ClientTravel**. Má dva způsoby volání. Pokud je volána serverem, tak způsobí, jak již bylo zmíněno výše, přenesení určitého hráče na novou mapu. Pokud je ovšem volána přímo klientem, způsobí jeho přenos na nový server. Na obrázku níže lze vidět volání funkce **UWorld::ServerTravel**, která byla použita při vytváření hry Bludiště, jejíž vytvoření bylo součástí mé práce.



Obrázek 18: Použití funkce UWorld::ServerTravel

Na obrázku 18 je možné vidět, že je tato funkce volána pomocí konzolového příkazu, kde se za samotným názvem funkce nachází cesta k dané mapě. Dále je zde vidět, že před samotným konzolovým příkazem je volána funkce **Switch on Int**, která potom volá jednu z následujících funkcí v závislosti na jejím vstupu, který v tomto případě je volba jedné z map.

Další krok, který je třeba provést, pokud chceme, aby přenášení fungovalo správně a hlavně pokud chceme, aby se jednalo o přenos typu **Seamless**, je nastavení takzvané přechodové mapy. Toto nastavení se provádí v **Project Settings** v sekci **Maps & Modes**. Implicitně je tato vlastnost nastavena na prázdnou hodnotu a pokud to tak zůstane tak si engine sám pro tyto účely vytvoří prázdnou mapu.

Hlavním důvodem proč něco jako přechodová mapa existuje je, že zde musí stále být načtená mapa, která je vázaná na hru, čili nelze pouze zahodit starou mapu a poté načíst novou. Samozřejmě by šlo udělat, aby stará mapa byla stále načtená během přenosu klientů, jenže v případě že je herní mapa obrovská, je velice nevýhodné udržovat v paměti, jak mapu původní, tak mapu novu. Čili výsledný přenos se prování z původní mapy na mapu přechodovou a z té poté na mapu novu. Jakmile je tedy přechodová mapa nastavena, musí se ve třídě **Game Mode** nastavit možnost **Use Seamless Travel** na **True**.

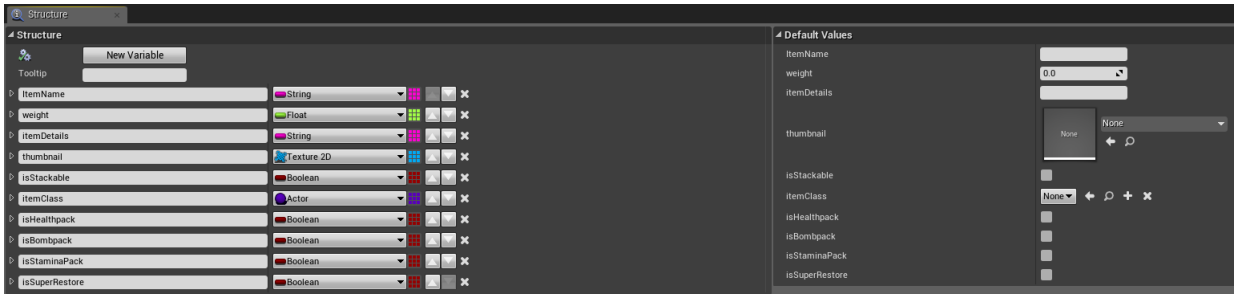
3 Inventář

Existuje spousta způsobů, jak zpracovat základní logiku inventáře, od inventáře založeného na maximální nosnosti hráče, po omezení počtu věcí, které hráč může u sebe mít. Ovšem všechny vycházejí z toho, že hráč musí být schopen daný předmět sebrat, vyčíst z něj potřebné informace a tyto informace graficky zobrazit ve svém inventáři a v neposlední řadě musí být schopen takto sebraný předmět použít. Tato kapitola je věnována popisu vytvoření inventáře s výše zmíněnými vlastnostmi, jak byl vytvořen pro účely této práce. Jelikož je práce zaměřena na hru více hráčů, tak i následující postup bude zaměřen na to, aby vše fungovalo po síti.

3.1 Vytvoření sběratelných předmětů

Než jsem vůbec mohl začít s vytvářením inventáře, tak jako první věc, kterou jsem si musel určit, byl způsob uchovávání informací o předmětech, které budou moci být uloženy v inventáři. Jeden z možných způsobů je samozřejmě vytvoření velkého množství proměnných, kde by každá uchovávala specifickou informaci o daném předmětu, jako například jméno, hmotnost nebo text sloužící k popisu daného předmětu. Mnohem přehlednější způsob je ovšem vytvoření si datové struktury, která všechny tyto proměnné bude v sobě zahrnovat a tedy pokud bude potřeba tyto proměnné někde používat nebo měnit, stačí se vždy odkázat na danou strukturu.

V UE4 je vytvoření takovéto struktury velice jednoduché. Nachází se v panelu **Blueprints** pod názvem **Strucure**. Jak je vidět na obrázku níže, přidávání proměnných do takovéto struktury a nastavování jejich výchozích hodnot není nijak složité.



Obrázek 19: Vytvoření struktury pro inventář

Na obrázku 19 lze vidět mnou vytvořená struktura, kde na levé straně se nachází proměnné a na straně pravé jejich výchozí hodnoty, které jsou v tomto případě prázdné, jelikož většina předmětů se bude v těchto proměnných lišit.

Po vytvoření struktury jsem tedy byl schopen ukládat potřebné informace a mohl jsem tedy přejít k samotnému vytváření předmětů. Stejně jako tomu je v mnoha programovacích jazycích, tak také v UE4 je možné vytvářet dědičnost tříd. Tuto vlastnost jsem využil při vytváření mých předmětů. Důvod tohoto rozhodnutí spočívá v tom, že pokud bychom měli například velký počet mečů, které by se rozdělovaly do vícero druhů a ty by se dále dělily podle určitých vlastností

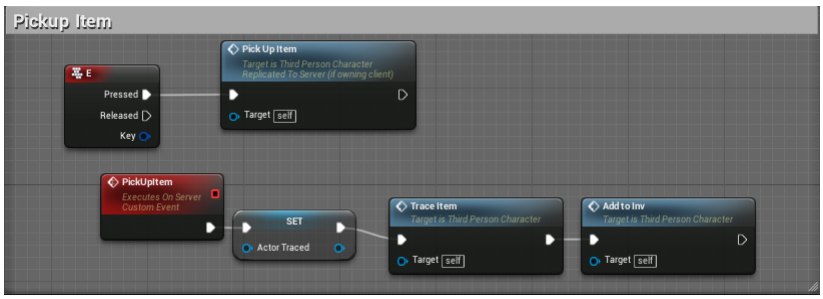
jako je například ostrost nebo výdrž a jiné, tak by bylo velice nevýhodné při vytváření dalšího takového meče, vytvořit zcela novou třídu pro tento druh meče a nastavovat zde jeho vlastnosti. Místo toho jsem si vytvořil základní třídu, kterou jsem pojmenoval **Item Master**, z nichž dědily všechny ostatní třídy předmětů, které pokud se dělily na různé druhy, tak tyto druhy poté dále dědily vlastnosti ze svých předchůdců. Toto mi zajistilo, že v případě, že daný předmět obsahovat část kódu, která spravovala jeho chování v určitých situacích, tak jsem již nemusel tento kód znovu implementovat ve třídě, která z třídy předchozí dědila. Děděním se samozřejmě s funkcemi přenášely také všechny proměnné, respektive již výše zmíněná struktura která v sobě veškeré proměnné uchovávala a stačilo v ní měnit pouze ty proměnné, které se liší vzhledem k rodiči dané třídy. Nakonec je třeba zmínit, že aby změny provedené na těchto předmětech byly viditelné pro všechny hráče připojené do hry, musí být u nich nastavena vlastnost **Replicates** na **true**.

3.2 Implementace třídy Player Controller

Tato třída již byla mnohokrát zmíněna v souvislosti s replikacemi a tudíž již zde nebude dále rozebírána. Pouze pro připomenutí se jedná o třídu, která obsahuje souhrn funkcí a vlastností, které má daný hráč, používající daný **PlayerController**. V mém případě to byla třída, ve které jsem musel vytvořit veškerou logiku sbírání předmětů a jejich následné umísťování do inventáře.

Jako první jsem si musel vytvořit proměnnou, která mi bude udržovat všechny předměty, které má daný hráč u sebe v inventáři. Nazval jsem si ji **Inventory** a jednalo se o pole struktur, které obsahovaly data o předmětu, respektive těch struktur jejíž vytvoření bylo popsáno v předcházející kapitole. Pro případné použití této proměnné po síti jsem ji nastavil na **replicated**. V případě, že nebude třeba, aby server a ostatní hráči věděli o obsahu inventáře dalších hráčů, může být tato vlastnost nastavena na **none**.

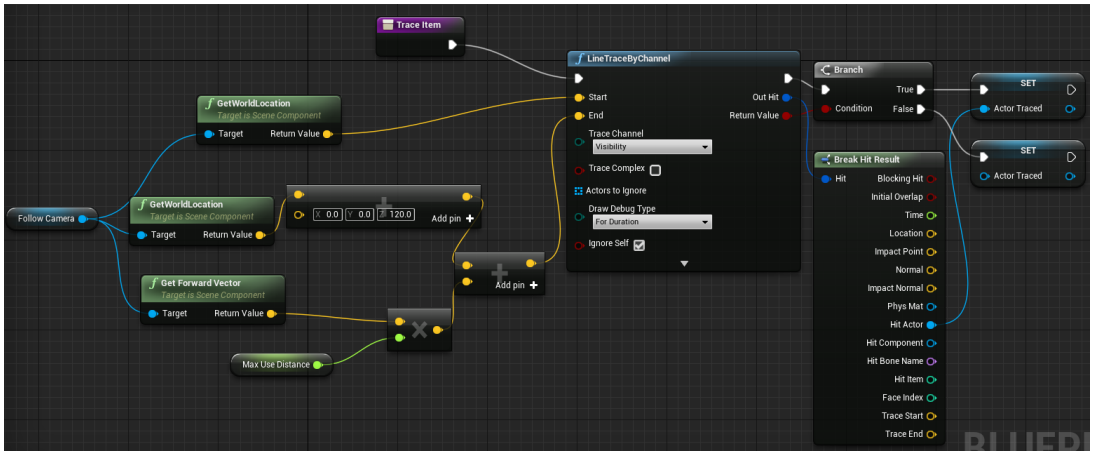
Následovala nutnost vytvoření funkce po naplnění této proměnné. Na následujícím obrázku lze vidět základní logika takovéto funkce.



Obrázek 20: Funkce pro sbírání předmětů

Na obrázku 20 lze vidět funkce **set**, která nastavuje hodnotu proměnné **ActorTraced**. Jedná se o proměnnou, kterou jsem si vytvořil pro uchování informace o předmětu, který jsem se právě pokusil sebrat a jehož informace budu chtít vkládat do proměnné **Inventory**. Pokud tedy kterýkoliv z hráčů zmáčkne klávesu E, je zavolána funkce **PickUpItem**, která nejprve vynuluje

proměnnou **ActorTraced**, aby nedocházelo k zapisování té samé proměnné stále dokola a poté zavolá funkci **TraceItem**. Tato funkce se stará o vystřelení paprsku z místa určení určitým směrem a kontroluje, zda tento paprsek na své cestě zasáhl nějaký objekt a pokud ano, uloží tento objekt do již zmíněné proměnné **ActorTraced**. Na obrázku níže lze vidět blueprint takové funkce.

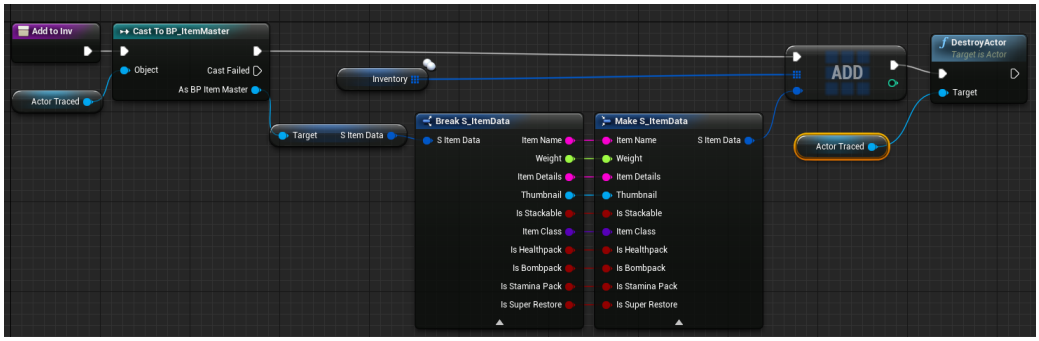


Obrázek 21: Funkce TraceItem

Obrázek 21 ukazuje, že funkce **LineTraceByChanel**, která se stará o vypuštění již zmíněného paprsku, požaduje dva povinné vstupy. Jsou jimi vstupy **Start** a **End**, neboli místo odkud je paprsek vysílán a místo kam paprsek následně dopadá. Obě tyto informace jsou v mém případě získávány z pozice kamery, která se v režimu **ThirdPersonGame** nachází za hráčem a tudíž její úhel odpovídá přibližně úhlu, kterým se hráč dívá. Jako start jsem si tedy určil přímo lokaci kamery a jako cíl jsem použil směrový vektor kamery, který jsem vynásobil proměnnou **MaxUseDistance**, která udává maximální možný dosah, ve kterém je hráč schopen sbírat předměty. Tento výsledek jsem přičetl k pozici kamery a následně zvedl o 120 po ose **z**, aby tento paprsek procházel přímo hlavou hráče a odpovídal tak skutečnému úhlu jeho pohledu. Dostal jsem tak místo, na které se hráč dívá a ze kterého se snaží sebrat předmět.

Tato funkce má také dva výstupy. Jsou jimi **OutHit** a **Return Value**. Druhý ze zmíněných výstupů mi vracel hodnotu **true** v případě, že paprsek zasáhl nějaký předmět a **false** pokud nezasáhl nic. Naproti tomu výstup **Out Hit** vrací strukturu, obsahující detailní popis předmětu, který byl zasažen. Obsahuje informace jako například jeho pozici ve světových souřadnicích, materiál a nebo přímo bod dopadu paprsku. V případě, že hodnota výstupu **Return Value** byla **true**, jsem tedy nastavil hodnotu proměnné **ActorTraced** na hodnotu jenž je obsažená ve zmíněné struktuře pod názvem **HitActor**. To mi umožnilo uložit si referenci na daný předmět, což jsem využil v následujícím kroku.

Na obrázku 20 následuje hned za funkcí **TraceItem** funkce **AddToInv**, která se již stará o uložení předmětu, který získala předchozí funkce do inventáře. Její kód je ukázán na následujícím obrázku.



Obrázek 22: Funkce AddtoInv

Funkce jako první ve svém těle volá funkci **Cast To** [14], která nejdříve kontroluje, zda je předmět uložený v proměnné **ActorTraced** stejného typu jako mnou vytvořená třída **ItemMaster**. Pokud ano, tak z toho předmětu vytáhne strukturu uchovávající informace o tomto předmětu a zapíše je do nové struktury, která se zde dočasně vytváří. Tuto strukturu následně vloží do pole struktur s názvem **Inventory** a nakonec daný předmět zničí, čímž způsobí jeho zmizení z herní mapy. V této funkci je jeden problém a to je samotné přepisování struktury do struktury nové, jelikož je nutné si pamatovat, že v případě, že bude potřeba předmětu přidat novou vlastnost a tedy novou proměnnou do jeho struktury, bude také třeba vrátit se do této funkce a propojit novou proměnnou ze struktury s proměnnou ve struktuře nové, jelikož UE4 toto automaticky neudělá a tato informace by se tedy do inventáře nepřenesla.

V této fázi jsem tedy měl vytvořené předměty, které mohl hráč sbírat, měl jsem k nim vytvořenou strukturu, která obsahovala veškeré informace a měl jsem také naimplementovanou funkci pro přidávání těchto předmětů do hráčova inventáře. Na řadě tedy bylo vytvoření samotného uživatelského rozhraní inventáře.

3.3 Vytvoření uživatelského rozhraní

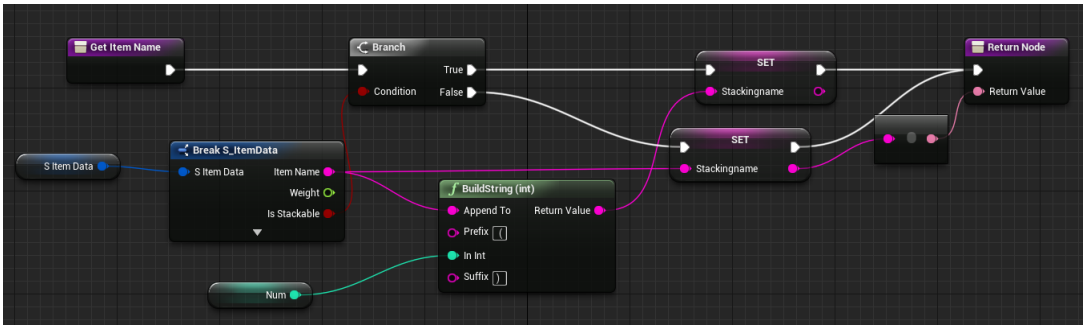
Z hlediska funkčnosti inventáře se sice jedná o nejméně důležitou část, která je velice subjektivní a ve své podstatě nijak neovlivňuje funkčnost inventáře, ovšem je třeba si určit, jak bude inventář vypadat a na jakém principu bude fungovat. Ve své práci jsem vytvářel inventář, jehož hlavní logika byla založena na maximální nosnosti hráče, čili každý předmět měl určitou hmotnost. Zvolil jsem také inventář, který pro vizualizaci jednotlivých položek používal pouze tlačítka s textem odpovídajícím názvu daného předmětu, kde po kliknutí na dané tlačítko došlo k zobrazení informací o daném předmětu a také k změně textu tlačítek, které se staraly o následné použití či zahození předmětu. Jak jsem již zmínil vzhled a funkčnost inventáře je zcela subjektivní záležitost, co ovšem tyto inventáře spojuje je možnost hráče sbírat předměty a následně je zobrazovat ve svém inventáři. Celé rozhraní inventáře bylo vytvořeno pomocí nástroje **Widget Blueprint** [10], respektive pomocí dvou **Widgetů**, kde jeden se staral o celkové zobrazení in-

ventáře a druhý pouze o zobrazení jednotlivých předmětů, které se v něm nacházely. Na obrázku níže lze vidět, jak takový mnou vytvořený inventář vypadal.

Obrázek 23: Inventář GUI

Na obrázku 23 lze vidět, že na levé straně se nachází hlavní část inventáře obsahující zmíněné tlačítka s názvy předmětů a jejich počtem. Po kliknutí na jedno z těch tlačítek se v pravém okně zobrazí informace o tomto předmětu. V pravém dolním rohu se poté nacházejí tlačítka pro následné operace s danými předměty a jako poslední zde lze vidět obrázek vybraného předmětu.

Co se týče samotného tlačítka, tak bylo zapotřebí naimplementovat dvě základní funkce, které by se staraly jednak o rozpoznání toho, na které tlačítko hráč klikl a jednak o přepsání textu tlačítka na jméno předmětu. Pro implementaci druhé ze zmíněných funkcí jsem použil možnost UE4 použít takzvaný **Binding**, který „sváže“ dohromady text na tlačítku s příslušnou funkcí. Výsledná funkce tedy vypadala následovně.



Obrázek 24: Funkce pro získání názvu předmětu

Jak lze vidět na obrázku 24, tak se jednalo pouze o vytažení proměnné **Item Name** ze struktury předmětu a následné přepsání textu tlačítka touto proměnnou. Je zde také vidět, že zde dochází ke kontrole zda je daný předmět možno stohovat, jelikož ne u všech předmětů je toto žádoucí. U předmětů jako například meč, který může mít vlastnost výdrž bude nutné, aby se tyto předměty zobrazovaly odděleně a bylo tak znát, který z mečů je který. Dále za každým názvem

se přidává číslo, které je zde reprezentováno proměnnou **Num**, která udává počet předmětů stejného druhu nacházející se v inventáři. Jak se tato proměnná získává bude vysvětleno níže.

Druhá funkce, která musela být neimplementována v rámci funkčnosti tlačítka, bylo samotné rozpoznání, na který předmět hráč klikl. Funkce po každém kliknutí hráče na tlačítko kontrolovala, na jakém místě v poli struktur se nachází struktura, která odpovídá vybranému předmětu a poté daný index uložila do proměnné **InvButtonClicked**. Tímto jsem získal přehled o hráči vybraném předmětu a vždy když jsem potřeboval vypsat informace o daném předmětu, jsem tedy vybral strukturu z daného pole nacházející se na indexu jenž odpovídal hodnotě proměnné **InvButtonClicked**.

Nyní jsem tedy měl vytvořenou logiku funkčnosti jednotlivých položek v inventáři a zbývalo naimplementovat jejich samotné zobrazení a případné operace, které s danými předměty mohl hráč provádět.

Ve **widgetu**, který se staral o celkový vzhled inventáře jsem tedy po zavolání funkce **Event Construct**, která je volána pokaždé, když je daný **widget** vytvořen, čili pokaždé když hráč otevře inventář, zavolal funkci, která mi pro každou položku v proměnné **Inventory** vytvořila již zmíněné tlačítko a toto tlačítko přidala do **scrollboxu**, který se nacházel uvnitř mého **widgetu** jak lze vidět v levé části obrázku 23. Než se ovšem k samotnému vytvoření tlačítka došlo, proběhla vždy kontrola, zda se položka v inventáři nenachází více než jednou a pokud ano, tak zda má vlastnost stohovatelnost nastavenou na **true** a pokud byly obě tyto podmínky splněny, tak se místo vytvoření nového tlačítka pouze zvětšila hodnota čísla nacházející se za daným předmětem o 1. Tím se vytvořila vizuální informace o počtu předmětů stejného typu nacházejících se v inventáři.

Jak už bylo zmíněno při popisu uživatelského rozhraní inventáře, tak jak je vidět na pravé straně obrázku 23, se nachází tabulka obsahující detailní popis předmětu, který byl právě vybrán. Změnu příslušných textů na text odpovídající danému předmětu jsem provedl stejně jako tomu bylo u změny textu na tlačítku a to pomocí takzvaného **bindingu**, kde jsem jednotlivé části textu nahradil hodnotami proměnných nacházejících se ve struktuře předmětu, která se nachází v poli struktur na indexu, který se mi po kliknutí na dané tlačítko uložil do proměnné **InvButtonClicked**. Tuto proměnnou jsem také využil při změně textu na posledních částech mého inventáře a to na tlačítkách starajících se o vyhození předmětu z inventáře a o jeho použití, kde vždy za slovem **Drop** nebo **Use** následovalo jméno vybraného předmětu.

Funkce **Drop Item**, která je volána po kliknutí na příslušné tlačítko, funguje na principu odstranění prvku z pole struktur na pozici, jejíž index odpovídá hodnotě proměnné **InvButtonClicked**. Po této funkci je zavoláno zavření inventáře a jeho následné otevření, což sice žádný z hráčů nepostřehne ovšem to způsobí zavolání již zmíněné funkce **Event Construct**, která znovu projede veškeré prvky pole **Inventory**, tentokrát ovšem bez prvku, který byl odstraněn. Jednoduše se tímto inventář aktualizuje. Posledním krokem této funkce je samotné vytvoření vyhozeného předmětu v oblasti kolem hráče, který předmět vyhodil. Je důležité zde zmínit, že tato část musí být volána na serveru, jelikož se jedná o přidání nového aktora do

scény, což v případě volání na straně klienta způsobí, že daný předmět nebude vidět nikde jinde, kromě daného klienta.

Druhá z funkcí je tedy funkce **Use Item**, která nejprve, stejně jako funkce předchozí, zkontroluje pomocí proměnné **InvButtonClicked**, který předmět byl vybrán a poté nejprve provede odpovídající funkci, například doplní hráči životy, pokud se jedná o předmět sloužící k tomuto účelu a následně předmět z inventáře odstraní.

4 Systém vyrábění věcí

Tento systém je velice úzce spjatý s inventářem, jelikož přímo využívá data, jenž inventář obsahuje. Jedná se tedy o systém, který na základě zvoleného receptu nabídne hráči přehled o tom, k čemu se výsledný předmět používá, jaké jsou požadavky na jeho výrobu a také hráči zobrazí, co vše má v inventáři, aby se mohl případně podívat, co vše mu chybí pro vyrobení zvoleného předmětu. Po splnění všech požadavků je tedy předmět vyroben a přidán do hráčova inventáře.

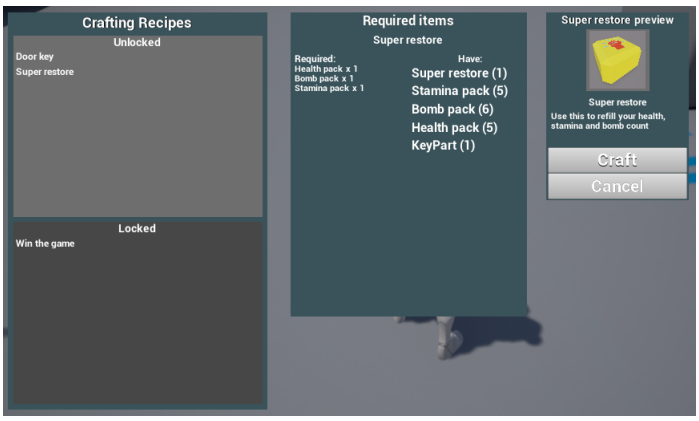
4.1 Vytvoření receptů

Stejně jako tomu bylo o vytváření sběratelných předmětů v předchozí kapitole, tak i zde jsem musel vytvořit strukturu, ve které jsem si vytvořil proměnné pro uchovávání důležitých informací o daném receptu. Na rozdíl od předmětů jsem zde kromě informací o typu objektu, jeho jméně a podobně, musel uchovávat informace o počtu všech druhů předmětů, které se daly použít k výrobě a také proměnnou, která o daném receptu říkala, zda je odemčený nebo uzamčený, jelikož jsem vytvářel systém, který hráči umožňoval přístup pouze k některým receptům a recepty zbylé mu byly odemčeny po splnění určitých požadavků.

Po vytvoření struktury byl postup opět velice podobný inventáři. Následovalo tedy vytvoření rodičovské třídy **RecipeMaster**, která obsahovala výše zmíněnou strukturu a ze které ostatní recepty dědily. Oproti inventáři jsem zde ovšem neprováděl žádnou implementaci ve třídě **Player Controller**, jelikož veškerá logika systému byla implementována v rámci třech **widgetů**, které se staraly o jeho zobrazení.

4.2 Rozhraní systému

Pro funkčnost celého systému jsem tedy zvolil spolupráci třech **widgetů**, které se podobně jako u inventáře staraly jednak o jednotlivé recepty a jednak o celkový vzhled rozhraní. Na obrázku níže lze vidět jak toto rozhraní vypadalo.



Obrázek 25: GUI Systému vyrábění věcí

Na levé straně obrázku 25 se nachází dva **scrollboxy**, kde každý z nich má jinou barvu pozadí. Jedná se o oddělení odemčených a neodemčených receptů. Každý z nich tedy obsahuje seznam **widgetů**, respektive tlačítek, které stejně jako tomu bylo u inventáře reprezentují jednotlivé recepty. Uprostřed se nachází část, sloužící k porovnávání potřebných předmětů s těmi, které má hráč v inventáři. V levé části se nachází opět seznam **widgetů**, které ovšem tentokrát neobsahují tlačítka, ale pouze text, který se skládá z názvu předmětu a jeho počtu. V pravé části je poté zobrazen inventář hráče, které zde ovšem slouží pouze pro informativní účely a tedy po kliknutí na dané předměty se nic nestane. Poslední část systému se nachází na pravé straně obrázku 25 a slouží k zobrazení výsledného předmětu, který po vyrobení vznikne a také samotné tlačítka pro výrobu a pro zrušení výroby.

Oproti inventáři zde byl ovšem jeden velice zásadní rozdíl. Do inventáře se předměty přidávaly po tom, co je hráč sebral a tedy až v té chvíli měl hráč přístup k informacím o tomto předmětu a mohl ho následně použít. V případě systému vyrábění věcí je ovšem potřeba, aby veškeré recepty a jejich informace byly dostupné již ze začátku. Hráč totiž musí mít přehled o tom, jaké recepty daná hra nabízí a co k tomu potřebuje. Ovšem stejně jako u každého programovacího jazyka, pokud chceme přistupovat například k poli čísel, musí toto pole nejdříve logicky existovat. Nabízí se tedy dva základní postupy, jak tento problém vyřešit. První způsob je vytvoření tlačítek s recepty s předem vytvořeným názvem, které po kliknutí provedou předem naimplementovanou funkci a sice zobrazení informací o daném receptu, jeho požadavcích a výsledku. Tento postup je ovšem velice neefektivní jelikož by se muselo při každém přidání nového receptu implementovat nové tlačítko a v případě, že by se tvůrce hry rozhodl nové recepty přidávat ve velkém množství a pravidelně, by se to stalo velice nepříjemnou součástí vývoje takové hry. Způsob, který jsem v rámci své práce zvolil je ten, že při spuštění hry se nejdříve vytvoří **blueprint RecipeMaster**, který se přidá do scény, čímž se všem hráčům otevře možnost přistupovat k jeho informacím, následně se z něj vytáhne proměnná **CraftingRecipeClasses**, která obsahuje referenci na všechny recepty, které z této třídy dědí a ty jsou následně také umístěny do scény, aby s nimi bylo možno pracovat. Tyto recepty spolu s jejich rodičovským **blueprintem** obsahují pouze data, čili jsou pro hráče neviditelné a nepřekáží tak ve hře. Důležité ovšem je, že existují jíž ze začátku hry.

Po získání informací o receptech jsem mohl přistoupit k vytvoření jednotlivých **widgetů**. Jako první bylo potřeba vytvořit **widget** obsahující informace o předmětech potřebných k výrobě vybraného receptu. Jak už bylo zmíněno tento **widget** obsahuje pouze název předmětu a jeho počet v textové podobě. Obě tyto informace byly nahrazeny hodnotami ze struktury receptu pomocí **bindingu**.

Druhým **widgetem** bylo již tlačítko s názvem daného receptu, který se následně bude přidávat do seznamu. I v tomto případě byl použit **binding** pro přepsání názvu tlačítka na název receptu a také zde byla naimplementována funkce velice podobná té, jenž byla popsána v souvislosti s inventářem a sice funkce, která byla zavolána po stisknutí daného tlačítka a která měla za úkol uložit číslo daného receptu do proměnné **Recipe Selected**, které odpovídalo jeho

pozici v poli receptů jako tomu bylo u proměnné **InvButtonClicked**. Následně po stisknutí tlačítka byla také zavolána funkce pro vytvoření seznamu potřebných předmětů pro výrobu zvoleného receptu. Tato funkce na začátku smazala veškeré informace nacházející se v seznamu potřebných předmětů, aby po opětovném stisknutí tlačítka nedocházelo k dvojímu zobrazení informací. Následně tato funkce nahlédla do struktury receptu, který měl v sobě proměnné obsahující informace o tom, jaké předměty je třeba mít v inventáři, aby bylo možné recept použít. Na základě těchto informací byl pro každý takový předmět vytvořen první z již výše zmíněných **widgetů**, který byl následně přidán do seznamu.

Zbývalo již tedy vytvořit poslední **widget** a to ten, který se bude starat o celkové zobrazení mnou vytvořeného systému vyrábění věcí. Opět je zde jistá podobnost s inventářem, jelikož i zde byla po otevření tohoto systému hráčem zavolána funkce **Event Construct**, která ve svém prvním kroku nejdříve vytvořila **widget** pro inventář. Inventář jsem zde musel vytvořit ze stejného důvodu, z jakého jsem musel umístit recepty do scény. Bylo potřeba přistupovat k jeho informacím a ty pak následně zobrazovat kvůli možnosti porovnávání potřebných věcí s věcmi v inventáři. Pokud jsem tedy zavolal vytvoření inventáře, zavolala se jeho funkce **Event Construct** a on tak obsahoval informaci o všech předmětech, které daný hráč měl. Samozřejmě, že po jeho vytvoření jsem musel nastavit jeho viditelnost na **hidden**, aby nešel ve hře vidět, a byl tedy pouze nosičem informací.

Po této funkci se dále v rámci **Event Construct** zavolala funkce pro vytvoření jednotlivých **widgetů** pro každý existující recept, a jeho následné přidání do jednoho ze seznamů v závislosti na tom, zda se jednalo o recept odemčený nebo uzamčený a nakonec se zavolala funkce pro vytvoření tlačítka pro každý předmět nacházející se v inventáři hráče a přidání tohoto tlačítka do příslušného seznamu.

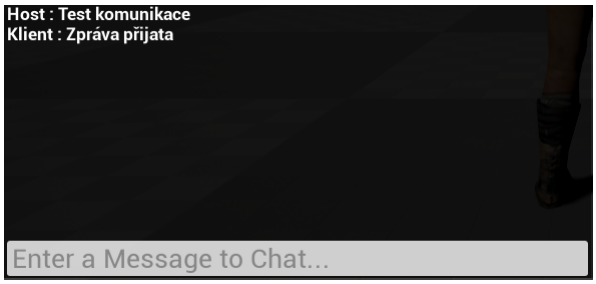
Posledním a nejdůležitějším krokem při vývoji systému vyrábění věcí byla implementace tlačítka **Craft**, které po tom, co na něj hráč klikl vyrobilo vybraný předmět a přidalo jej do hráčova inventáře. Tato funkce si nejdříve do předem vytvořených proměnných uložila počet předmětů, které hráč má ve svém inventáři a poté tento počet porovnala s hodnotou příslušné proměnné, nacházející se ve struktuře receptu. Pokud byla jejich hodnota větší nebo rovna hodnotám v dané struktuře a pokud tedy hráč vlastnil veškeré potřebné předměty pro výrobu, byl tento předmět vyroben a umístěn do inventáře. Následně byla zavolána funkce, která předměty, jenž byly na výrobu využity, z inventáře odstranila.

5 Komunikace v sítové hře

Jedním z cílů mé práce bylo také vytvoření komunikace mezi jednotlivými hráči připojenými do hry. Velkým rozdílem oproti předešlým kapitolám bylo to, že zde bylo opravdu nutné, aby veškerá logika sloužící pro odesílání zpráv pracovala na serveru a tudíž její výsledek byl dostupný pro všechny připojené hráče. Základní struktura komunikace spočívala ve vytvoření takzvaného **chatu**, skrze něj by hráči mohli komunikovat.

5.1 Implementace uživatelského rozhraní

Jednou z výhod implementace komunikace po síti bylo, že většina logiky starající se o posílání zpráv se dala naimplementovat přímo v rámci uživatelského rozhraní a tudíž v rámci příslušných **widgetů**, které pro tuto potřebu byly vytvořeny dva. Jediný nutný zásah do tříd typu **Player Controller** spočíval v zavolání aktualizací zmíněných **widgetů** nejdříve na serveru a následně na každém z klientů. Na obrázku níže je vidět, jak takové rozhraní pro chat vypadá.

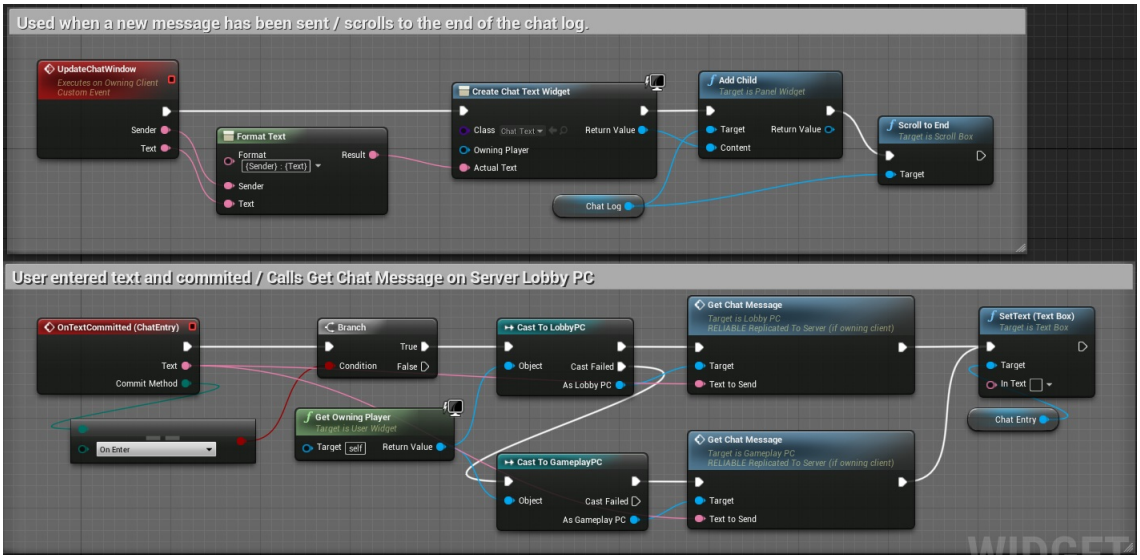


Obrázek 26: GUI pro komunikaci po síti

Na obrázku 26 je možné vidět jistou podobnost mezi inventářem a systémem vyrábění věcí, jelikož i zde se jedná o přidávání **widgetů**, jenž tvoří jednotlivé zprávy, do výsledného **widgetu**, který se již stará o celkový vzhled. Stejně jako tomu bylo tedy u vytváření jednotlivých tlačítek pro předměty, tak i zde jsem si nejdříve musel vytvořit **widget** pro jednotlivé zprávy.

Jednalo se o zcela triviální postup, jelikož tento **widget** obsahoval pouze text, který byl následně pomocí **bindingu** nahrazen jménem odesílatele a zprávou, kterou poslal.

Následovala implementace hlavního **widgetu**, který měl za úkol tyto zprávy zobrazovat všem klientům. Hlavní logika spočívala v tom, že pokaždé když hráč napsal nějaký text a zmáčkl klávesu enter, zavolala se funkce, která v jeho třídě **Player Controller** nejdříve uložila informace o zprávě na serveru, což umožnilo její rozeslání mezi ostatní klienty a následné zavolání funkce pro aktualizace okna s chatem. Na dalším obrázku je možné vidět funkce naimplementované v rámci hlavního **widgetu**.

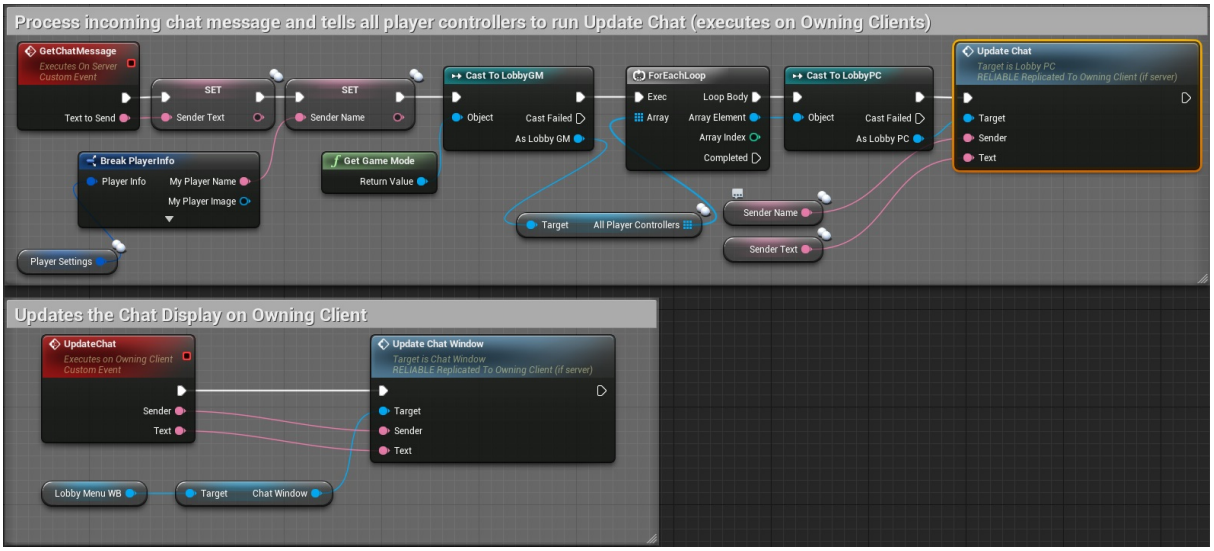


Obrázek 27: Implementace posílání zpráv v rámci hlavního widgetu

Jedna z funkcí na obrázku 27 je funkce **OnTextCommitted**, která je v tomto případě vázána na proměnnou **ChatEntry**, jenž odpovídá místu pro psaní zprávy (**text box**). Tato funkce je volána pokaždé když hráč píšící zprávu zmáčkne určitou klávesu, která je zde reprezentována proměnnou **Commit Method**. Na začátku tato funkce kontroluje zda daná klávesa, kterou hráč zmáčkl je právě klávesa enter. V případě, že tomu tak je, je volána funkce **Cast To**, která kontroluje zda daný hráč je ovládán jedním z uvedených **Player Controllerů**. Jelikož hra bludiště, ve které byl tento chat použit obsahuje dvě mapy, kde v každé byl hráči přidělen jiný **Player Controller**, musel jsem zajistit, aby tyto funkce fungovaly na obou mapách. Následně je tedy v rámci vybraného **Player Controlleru** volána funkce **GetChatMessage**, která jako vstup přímá text z již zmíněného **text boxu**. Posledním krokem je vynulování textu obsaženém v **text boxu**.

Druhá funkce **UpdateChatWindow** je volána na straně klienta a jedná se o samotné zobrazení zprávy. V prvním kroku funkce vytvoří **widget** se zprávou a předá mu jméno odesílatele a text zprávy. To poté nahradí proměnnou **Actual Text**, která jak již bylo zmíněno, změní pomocí **bindingu** text v daném **widgetu**. Následně je volána funkce **Add Child**, která tuto zprávu přidá do **scroll boxu** a tím ji danému klientovi zobrazí. Nakonec je zavolána funkce **Scroll to End**, která způsobí, že další zpráva se zapíše opět ve spodní části **scroll boxu**.

Nyní bylo samozřejmě třeba naimplementovat, aby se funkce **UpdateChatWindow** volala v rámci funkce **GetChatMessage**, čili pokaždé když hráč napíše zprávu a také bylo potřeba zajistit, aby se funkce **UpdateChatWindow** volala na všech klientech a ne pouze na tom, který ji vyvolá. Tato část logiky se již řešila ve třídě **Player Controller**. Na dalším obrázku je možné vidět, jak implementace v této třídě vypadala.



Obrázek 28: Implementace posílání zpráv v rámci třídy Player Controller

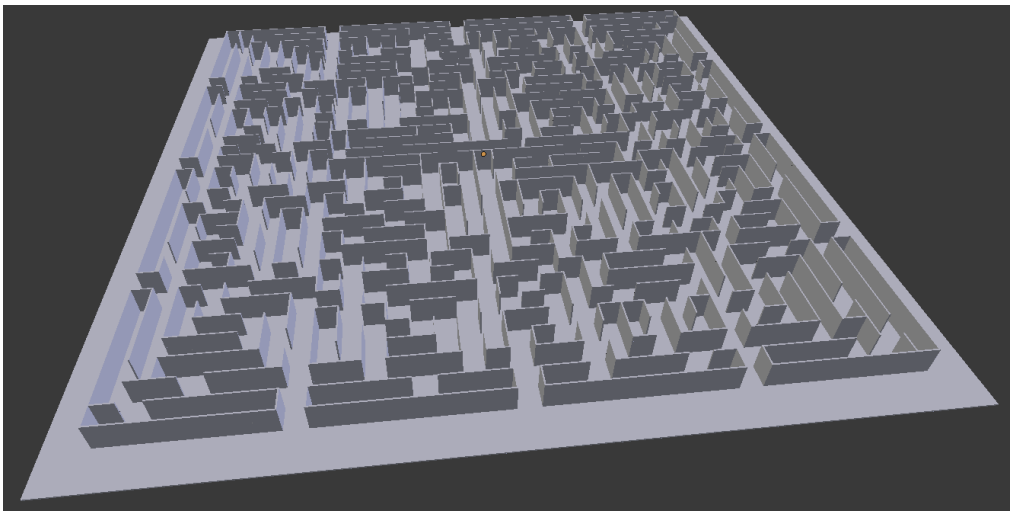
První ze dvou funkcí nacházejících se na obrázku 28 je již výše zmíněná funkce **GetChatMessage**, která má zde v třídě **Player Controller** svou implementaci a v rámci **widgetu** starajícího se o chat byla pouze volána. Jak lze vidět i na obrázku 27, tak má funkce vstup **Text to Send**, který je v rámci **widgetu nahrazen** textem z **text boxu** a následně zde ve třídě **Player Controller** se tento text uloží do proměnné **Sender Text**, která reprezentuje napsanou zprávu. Poté se ze struktury obsahující informace o hráči vytáhne jeho jméno a to se poté uloží do další proměnné s názvem **Sender Name**. Následuje získání pole **All Player Controllers** ze třídy **Game Mode**. Toto pole v sobě, jak už název napovídá, obsahuje třídy typu **Player Controller** všech připojených hráčů. Poté se pomocí cyklu **For Each** zavolá pro každý takový **Player Controller**, čili pro každého připojeného hráče funkce **Update Chat**, která si jako vstup vezme již zmíněné proměnné obsahující informace o zprávě a dále pak sama zavolá funkcí **Update Chat Window** z hlavního **widgetu** pro chat a tyto informace ji předá. Jak již bylo zmíněno tak funkce **Update Chat Window** aktualizuje na daném klientovi okno s chatem a právě její volání v rámci **For Each** cyklu zajistí, že je volána pro každého hráče a tedy všichni hráči připojení do hry budou vidět vše, co ostatní píšou.

6 Hra Bludiště

Hlavním cílem mé práce bylo využít získané vlastnosti o tvorbě síťové hry a následně je implementovat v rámci zadaného projektu, což v tomto případě byla právě hra bludiště. Hra byla založena na zkoumání chování lidí, jenž se v bludišti pohybují. Během hraní hry se ukládaly potřebné informace o každém hráči do předem vytvořených polí a ty se po ukončení hry zapisovaly do souborů, které ve svém názvu obsahovaly jména hráčů, kterým patřily.

6.1 Model bludiště

Jelikož cílem hry nebylo, aby se hráč dostal z jednoho bodu bludiště na druhý a našel východ, jako tomu většinou bývá, nýbrž jeho cílem bylo sbírat předměty, které se v bludišti nacházely a zabíjet ostatní hráče, bylo i bludiště navrženo právě pro tyto účely. Základním požadavkem bylo, aby mělo několik vstupů a hráči se náhodně objevovali u jednoho z nich. Při návrhu celkového modelu jsem se také zaměřil na vytvoření co nejvíce slepých cest a eliminaci dlouhých rovin pro co možná nejobtížnější průchod bludištěm. Poslední důležitou informací je rozloha bludiště. Pro účely mé práce byly vytvořeny bludiště dvou velikostí, přičemž obě byly čtvercového tvaru. První z nich mělo délku strany 99 metrů, kde šířka jednotlivých chodeb byla 3 metry. Druhé bludiště mělo délku strany dvakrát delší, ovšem nikdy nebylo použito za účelem testování chování lidí. Na dalším obrázku je zobrazen model prvního z bludišť, jak byl vymodelován v prostředí **Blender**[15].

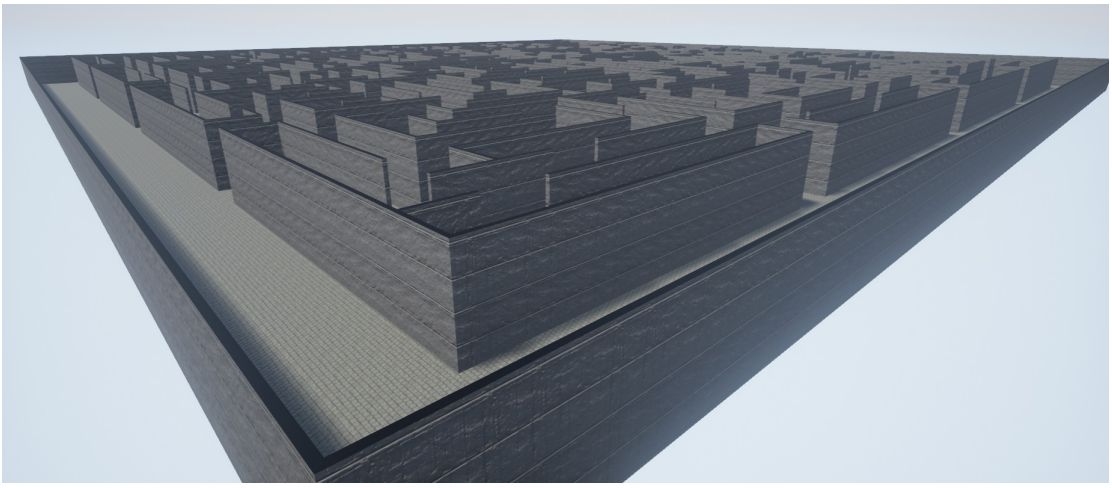


Obrázek 29: Model bludiště

Celkové modelování neprobíhalo formou vytváření jednoho velkého modelu, ale skládáním celého bludiště z jednotlivých zdí, jejichž délka odpovídala šířce chodby, čili se jednalo o 3 metry. Dále je na obrázku 29 vidět, že každá ze stran bludiště má 3 samostatné vchody, což ve výsledku

umožnilo snížení šance, že se dva hráči objeví na stejné pozici. Co se týče kolizních objektů, tak ty byly také vytvářeny v **Blenderu**.

Předposledním krokem bylo vytvoření materiálu a nanesení textury. Jelikož bylo bludiště tvořeno jednotlivými zdmi, tak stačilo nanést texturu na první zeď a tu poté pouze nakopírovat. Nakonec bylo bludiště naimporotváno do prostředí Unreal Engine 4, kde byly provedeny poslední úpravy materiálů. Byly zde přidány normálové textury a také došlo ke zvýraznění efektu lesku materiálu. Na obrázku níže je zobrazeno bludiště v prostředí Unreal Engine 4.



Obrázek 30: Bludiště v prostředí UE4

6.2 Vytvoření ukládání informací o hráči

V tomto případě ještě není myšleno samotné ukládání dat do souboru, které byly následně využity při zkoumání chování jednotlivých hráčů, ale pouze vytvoření struktury pro hráče, jako tomu bylo například u vytváření předmětů v inventáři, pro snadnější přístup k jeho informacím. Základní logika spočívala v tom, že si hráč zvolil své jméno, avatara a postavu, za kterou bude hrát. Byla tedy vytvořena struktura uchovávající tyto informace.

Tato struktura byla následně využita při vytváření jednoho typu z **blueprintů**, které UE4 nabízí a sice **SaveGame**. Tento **blueprint** slouží k vytvoření save souboru, který v sobě bude uchovávat proměnné, které mu byly předány a tyto proměnné poté mohou být v případě potřeby znovu načteny. V mém případě byla tomuto **blueprintu** předána struktura s informacemi o hráči. Využití tohoto ukládání informací bude vysvětleno v nadcházejících kapitolách.

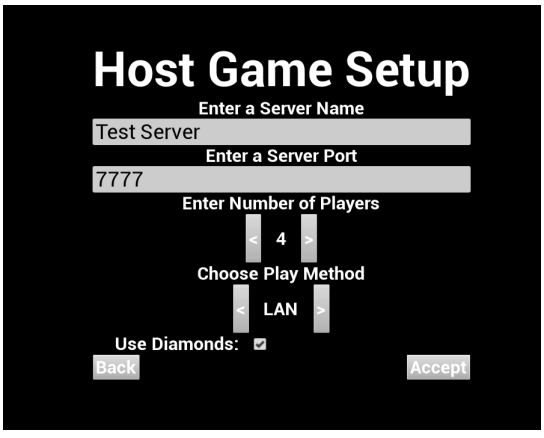
6.3 Implementace hlavního menu

Jedná se o základ každé hry, kde po jejím spuštění se hráči nabídne řada základních možností. V případě mé hry se jednalo o hostování hry, připojení se k vytvořené hře nebo nastavení svého profilu, respektive nastavení jména a avatara. A v tomto případě přichází na scénu již zmíněný **SaveGame**. Jelikož vždy, když hráč zadal své jméno a avatara, přepsala se hodnota těchto

proměnných v jeho struktuře hodnotami novými a proběhlo také přepsání jeho save souboru. Po každém zapnutí hry se poté kontrolovalo, zda daný soubor v počítači hráče existuje a pokud ne, místo hlavního menu se hráči zobrazilo přímo menu s nastavením jeho postavy, aby nedošlo k připojení do hry hráče, který by tyto vlastnosti nastavené neměl nebo na jejich nastavení zapomněl. Jak mnou vytvořené menu vypadalo je zobrazeno na následujícím obrázku.

Obrázek 31: Hlavní menu

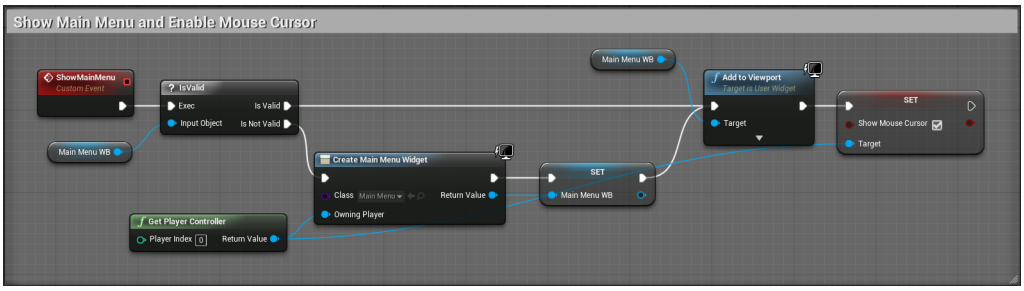
Po kliknutí na tlačítko **Host Game** se hráči nabídla možnost vytvoření nové hry, kterou by on sám hostoval na svém počítači. Nabídla se mu také možnost nastavit základní parametry serveru jako například jméno, maximální počet hráčů, kteří se na server mohou připojit a další. Důležité je zde zmínit, že hostování hry nejprve probíhalo na principu, který byl popsán a vysvětlen v rámci kapitoly **Použití Online Session Nodes** čili voláním funkcí **Create Session** a **JoinSession**, nicméně kvůli úrovní zabezpečení školní sítě, v rámci které testování chování lidí probíhalo, nebylo možné nakonec tyto funkce použít, jelikož byly blokovány a funkce **Find Session** nevracela žádné výsledky, ke kterým by se hráč mohl připojit. Byla tedy nakonec použita pouze funkce **Create Session** pro vytvoření dané **session**, ovšem hráči, jenž se chtěli na daný server připojit, museli u sebe zadávat IP adresu počítače, který hru hostoval, čili byl zde použit pouze konzolový příkaz **Execute Console Command**, tak jak je popsán v kapitole **Hostování pomocí konzolových příkazů**. Na dalším obrázku je znázorněno menu pro hostování hry.



Obrázek 32: Menu pro hostování hry

Položka **Use Diamonds** na obrázku 32 znamená, zda se v bludišti budou nacházet speciální předměty (diamanty), které jsou hodnoceny více body než základní mince. Tato možnost zde byla přidána za účelem pozorování změny chování lidí v případě, že se diamanty v bludišti nacházejí a nebo naopak nenacházejí.

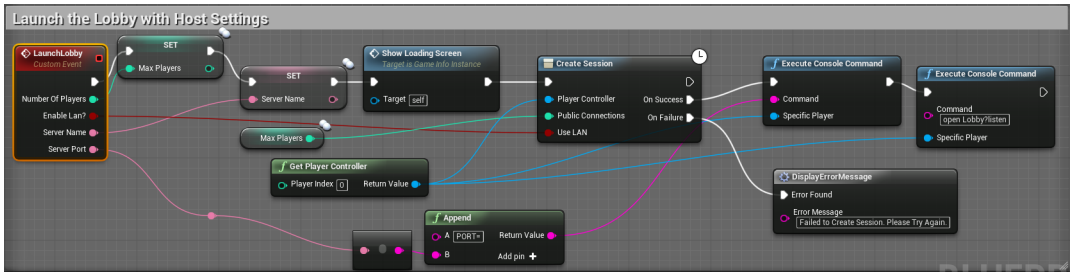
Aby bylo možné docílit jednoduchého přístupu k funkcím starajícím se o zobrazování jednotlivých složek hlavního menu, byla veškerá logika vytváření těchto **widgetů** naimplementována ve třídě **Game Instance**, která jak již bylo zmíněno, obsahuje veškeré funkce a proměnné, ke kterým chceme přistupovat z jakékoliv herní mapy. Jedná se také o třídu, ve které byly naimplementovány veškeré funkce starající se o správu případných chybových hlášek. Jelikož z hlediska implementace si jsou jednotlivé funkce pro zobrazování **widgetů** velice podobné, je na následujícím obrázku zobrazena pouze jedna vybraná a to funkce na zobrazení **widgetu** hlavního menu.



Obrázek 33: Příklad implementace zobrazení widgetu

Funkce na obrázku 33 byla, jak už bylo zmíněno, volána po spuštění hry v případě, že hráč již měl vytvořený save soubor s potřebnými informacemi. Funkce nejdříve vytvoří samotný **widget** a poté jej pomocí funkce **Add to Viewport** zobrazí hráči na obrazovku. Nakonec funkce zobrazí hráči kurzor myši, aby pomocí něj mohl menu ovládat.

Kromě již zmíněných funkcí, třída **Game Instance** také obsahuje veškerou logiku hostování a připojení hráče do hry. Pokud tedy hráč zakládající novou hru zmáčkne tlačítko **Accept** ve **widgetu** hlavního menu, zavolá se funkce **Launch Lobby**. Jak už bylo zmíněno hostování hry je zde provedeno pomocí funkce **Create Session**. Na dalším obrázku je celá funkce starající se o vytvoření nové **session** zobrazena.



Obrázek 34: Funkce pro hostování hry

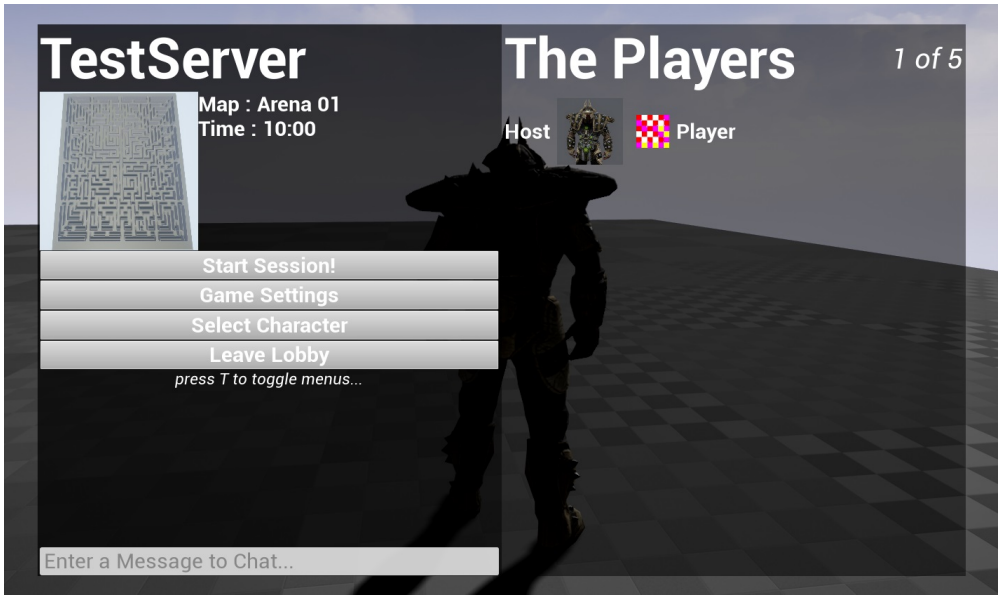
Funkce **Launch Lobby** z obrázku 35 požaduje při svém volání 4 vstupy. Jejich hodnoty získá z **widgetu** pro hostování hry, který byl zobrazen na obrázku 32. Poté si tyto hodnoty uloží do třídních proměnných pro pozdější použití a zavolá funkci **Show Loading Screen**. Tato funkce zobrazuje pouze **widget** s textem obsahujícím nápis „Loading...“, který po chvíli zmizí a nahradí jej načtená mapa s názvem **Lobby**, jak už vypovídá z následného konzolového příkazu. Pro svou práci jsem ještě před tím, než budou hráči vpuštěni do bludiště, zvolil přechodovou mapu, ve které si hráči budou moci vybrat jednu z nabízených postav a také je to mapa, kde hostující hráč může upravit poslední nastavení hry před jejím spuštěním. Hlavní výhoda této přechodové mapy je, že po spuštění samotné hry budou všichni hráči vpuštěni do bludiště ve stejnou dobu a ne v takovém pořadí, v jakém se připojovali.

6.4 Vytváření přechodové mapy

Od chvíle, kdy se hráč připojí na novou mapu s názvem **Lobby** mu bude přidělen nový **Player Controller**, obsahující všechny důležité funkce, týkající se jeho pobytu v této mapě. Jako první se přihlásí třída **Game mode**. Tato třída se vytváří vždy jako první a obsahuje tedy funkce, které je třeba provést ještě před tím, než hráč začne svůj **Player Controller** využívat.

Unreal Engine 4 nabízí funkci s názvem **Event OnPostLogin**, která se zavolá pokaždé když se nový hráč připojí do mapy, která je ovládána pomocí třídy **Game Mode**, ve které je tato funkce naimplementována. Tuto funkci jsem využil pro zavolání všech potřebných funkcí pro nově připojeného hráče, aby měl všechny důležité informace o hře. Nejdříve jsem si vytvořil pole **Player Controllerů**, do kterého jsem pokaždé když byla funkce **Event OnPostLogin** zavolána přidal příslušný **PlayerController** nového hráče, který daná funkce vrací na svém výstupu. Tímto způsobem jsem po připojení všech hráčů měl proměnnou, která mi uchovávala jejich **PlayerControllery** a já jsem se na ně mohl kdykoliv odkázat. Dále tato funkce ve svém těle volala funkci **Initial Setup**, která zkontrolovala danému hráči jeho save soubor, zda byl správně uložen a v případě, že nebyl ho pro něj vytvořila. K dalším funkcím, které jsou volány v rámci funkce **Initial Setup** se vrátím později. Následovalo vytvoření **widgetu** starajícího se o zobrazení hlavního menu, které bude hráč v této mapě využívat. Jeho zobrazení jsem provedl stejně jako je tomu na obrázku 33. Potom co bylo hráči menu zobrazeno jsem musel zajistit, aby informace jenž poskytuje byly aktuální. Zavolal jsem tedy funkci, která si v třídě **PlayerController** vytvořila proměnné obsahující například název zvolené mapy, její obrázek a podobně. Hodnoty těchto proměnných ji byly přiděleny třídou **Game Mode**. Posledním krokem funkce **Event OnPOstLogin** bylo samotné umístění hráče do mapy na jednu z náhodně vybraných pozic.

Na následujícím obrázku je zobrazeno hlavní menu pro mapu **Lobby** z pohledu hostujícího hráče. Menu vždy při svém vytvoření, tedy vždy když se zavolala funkce **Event Construct** zkontrolovalo, zda daný hráč je hostem nebo klientem a podle toho mu buď zobrazilo nebo zakrylo určité tlačítka pro funkce, které může vidět pouze hostující hráč, jako například podrobnější nastavení hry nebo její samotné spuštění.



Obrázek 35: Hlavní menu pro mapu Lobby

Na obrázku 35 se v levém spodním rohu nachází okno pro komunikaci po síťové hře, které bylo vytvořeno tak, jak bylo popsáno ve stejnojmenné kapitole. Přímo nad tímto oknem se poté nachází tlačítko **Leave Lobby**. Toto tlačítko se postará o zavolání funkce **Destroy Session** pro daného hráče a o následné zobrazení hlavního menu. Jednodušeji řečeno se jedná o nástroj pro opuštění hry a návratu k hlavním možnostem. V tomto případě je plně využita třída **Game Instance**, jelikož se pro zobrazení menu volá stejná funkce, jako při spuštění celé hry, jelikož nám tato třída umožnila její volání z kterékoliv mapy. Další tlačítko nese nápis **Select Character** a po jeho kliknutí se zobrazí **widget** s výběrem jedné z osmi postav, které jsou ve hře k dispozici. Veškeré postavy, které jsem ve hře použil, byly staženy v rámci jednoho z balíku, jenž UE4 nabízí. Pokaždé když hráč zaklikne jednu z nabízených postav, mu je přepsána hodnota proměnné v jeho struktuře, která se stará právě o uchovávání reference na hráčovu zvolenou postavu. Následně se taky zavolá funkce pro aktualizaci hráčova save souboru. Předposledním tlačítkem je tlačítko **Game Settings** a jak už bylo zmíněno, zobrazuje se pouze v případě, že daný hráč je hostem. Toto tlačítko nabízí hostovi možnost změnit mapu, na které se bude hrát a také vyhodit hráče ze hry tím, že na jeho straně zavolá již zmíněnou funkci **Destroy Session**. Posledním tlačítkem je **Start Session**. Toto tlačítko se zobrazuje jak hostovi, tak klientovi ovšem pokaždé s jiným textem a funkcí. Pokud je hráčem klient, tak tlačítko nese název **Toggle Ready** a umožní klientovi oznámit hostujícímu hráči, že je připraven na spuštění hry, jelikož v případě, že na toto tlačítko klikne se mu přepíše proměnná v jeho struktuře uchovávající jeho momentální status na „Raedy“. Tato proměnné je poté dále využívána, jak bude popsáno níže. Pokud je ovšem hráčem host, tak toto tlačítko způsobí přenesení všech hráčů do bludiště, ovšem je aktivní pouze v případě, že všichni hráči mají zvolenou postavu. Pokaždé když se do hry připojí nový hráč nebo si některý z hráčů změní postavu, je zavolána funkce, která projede struktury všech hráčů

a kontroluje, zda zde mají referenci na jednu z možných postav a pokud ano, je tlačítko pro spuštění hry zpřístupněno.

Nad těmito tlačítky se poté nachází informace o mapě a název serveru. Je důležité zmínit, že stejně jako celá struktura uchovávající informace o hráči, tak i proměnné nesoucí název mapy a serveru musí být nastaveny na **Replicated**, jelikož je nutné je zobrazovat všem hráčům. Pokaždé tedy když hostující hráč jednu z těchto proměnných změní, například změnou mapy, na které se bude hrát, se na každém klientovi zavolá funkce, která jim dané menu aktualizuje s novými hodnotami proměnných, ke kterým díky jejich nastavení na **Replicated** mají nyní přístup.

Na pravé straně obrázku 35 se poté nachází seznam připojených hráčů, který je tvořen jednotlivými **widgety**, jako tomu bylo například u seznamu předmětů v inventáři. Tento **widget** obsahuje základní informace o daném hráči a tyto informace jsou aktualizovány pokaždé, pokud je některé z těchto informací změněna, například změnou postavy hráče. Aby bylo ovšem možné o těchto změnách informovat i ostatní hráče ve hře, bylo třeba naimplementovat řadu funkcí, které tyto aktualizace prováděly automaticky. Tímto bych se rád vrátil k funkci **Initial Setup**, která je volána v rámci funkce **Event OnPostLogin**. Jak už bylo řečeno, tak tato funkce je volána pokaždé, když dojde k připojení nového hráče a jako první věc, kterou udělá, je kontrola hráčova save souboru. Následuje funkce **Call Update**, která jak název napovídá, má za úkol zavolat všechny potřebné funkce pro aktualizaci hlavního menu v mapě **Lobby**. Důležité je zmínit, že bylo potřeba tuto funkci nastavit na **Run On Server**, jelikož ve svém těle volala funkci **Swap Character**, která hráči po tom co se připojil, přiřadila jeho výchozí postavu a tu poté umístila do hry a jelikož se jedná o vytvoření nového aktora ve scéně, tak tato funkce musí být volána právě serverem. Poté co byla hráči přiřazena postava se zavolala funkce **Everyone Update**, která nejdříve inkrementovala hodnotu udávající počet hráčů, která je zobrazena na obrázku 35 v pravém horním rohu a poté pro každého připojeného hráče, čili za použití cyklu **For each** nad polem **Player Controlleru** jehož naplnění bylo popsáno výše, zavolala funkci **Add Player Info**, která byla již volána na straně každého z klientů a nejdříve vyčistila seznam připojených hráčů a poté pro každého připojeného klienta vytvořila **widget** s jeho informacemi a přidala jej do seznamu. Což způsobilo aktualizaci seznamu připojených hráčů s nejaktuálnějšími informacemi. Funkce **Call Update** nebyla volána pouze v rámci funkce **Initial Setup**, nýbrž pokaždé když hráč provedl jakoukoliv změnu ve svých základních informacích.

Po připojení všech hráčů, vybrání jejich postav a aktualizaci jejich informací, bylo možné ze strany hostujícího hráče hru spustit. Zde přichází největší uplatnění vytváření save souboru pro každého hráče. Když totiž hostující hráč spustil danou hru, došlo nejprve k zobrazení již zmíněného **widgetu**, který obsahoval pouze text „Loading....“ a následně zavolání funkce, která je zobrazena na obrázku 18 a jejíž funkce je také popsána v příslušné kapitole, která provede finální přenesení všech hráčů do hry. Ovšem samotné data o hráči se kvůli změně jeho základních tříd nepřenesou.

6.5 Implementace herní logiky

Jelikož jsem potřeboval, aby hráči měli jiné možnosti v mapě s bludištěm, než měli v předchozí mapě, vytvořil jsem novou třídu **Player Controller**, která se k mapě bludiště vztahovala. Stejně tak třída **Game Mode** byla nahrazená třídou novu. Jednoduché vysvětlení toho, proč jsem nemohl použít třídy minulé je to, že například v mapě bludiště jsou hráči schopni házet bomby, jejichž logika je naimplementována právě ve třídě **Player Controller** a já jsem samozřejmě nechtěl, aby tuto možnost házení bomby mohli hráči v přechodové mapě **Lobby** využívat.

Musel jsem ovšem zajistit, aby všechny informace o hráči, jenž si v sobě předchozí **Player Controller** uložil, byly přeneseny i na novou mapu a tedy do nových tříd. UE4 nabízí funkci s názvem **Event OnSwapPlayerControllers**, která je volána pokaždé, když je hráči přiřazen nový **Player Controller**. Tato funkce má dva výstupy, které reprezentují nový a starý **Player Controller**. Využil jsem toho k vytvoření si pole těchto tříd, jako jsem to udělal v předchozí mapě a tudiž pokaždé když byl přenesen hráč, se tato funkce zavolala a jeho nový **Player Controller** se mi uložil do mého pole. Měl jsem tex přehled o všech připojených hráčích do hry.

Jakmile byl nový **Player Controller** vytvořen a přidělen hráči, zavolala se v něm funkce s názvem **Event BeginPlay**, která je UE4 volána pokaždé, když je spuštěna hra, respektive když je spuštěná mapa, ve které má hráč přiřazený daný **Player Controller**. V rámci dané funkce se nejdříve zavolalo načtení hráčova save souboru, což umožnilo jeho nové třídě načíst si veškerá data, jenž si hráč nastavil v předešlých mapách a mít tak o hráči přehled. Nyní bylo třeba hráče umístit do hry a vytvořit mu jeho uživatelské rozhraní neboli **HUD**, který se bude starat o vizuální zobrazení veškerých herních informací, které bude hráč během hry využívat. Opět je zde nutné zdůraznit, že funkce, která se o vše výše zmíněné starala, musela být volána na serveru, protože umístění hráče do hry, by jinak nebylo pro ostatní viditelné. Nejdříve se tedy v rámci této funkce umístil hráč do hry k jednomu z náhodně vybraných vstupů do bludiště. Ze save souboru měla funkce informaci o jeho jméně a hlavně o jeho postavě, kterou si v minulé mapě vybral, což využila při jeho umísťování do hry, kde jeho výchozí postava byla nahrazena právě tou, jejíž referenci měl uloženou ve své struktuře. Následovalo vytvoření jeho **widgetu** který obsahoval jeho jméno, skóre a počet životů a byl umístěn v levém horním rohu obrazovky. Dále byl vytvořen seznam všech připojených hráčů obsahující jejich jméno a skóre a nakonec herní mapa. Poslední dva zmíněné **widgety** měly ze začátku nastavenou viditelnost na **false**, která se poté dala změnit kliknutím na příslušná tlačítka. Co se týče seznamu hráčů, byl vytvořen stejně jako předchozí seznamy v minulých kapitolách a aktualizován při jakékoliv změně některé z informací. Mapa byla vytvořena pomocí komponenty **Scene Capture2D**, kde nad herní mapu byla umístěna kamera, která ukládala snímky jenž pořizovala do textury, která byla následně hráči zobrazována v rámci zmíněného **widgetu**. Na dalším obrázku je možné vidět jak vypadalo umístění do hry z pohledu hráče.

Obrázek 36: Ukázka ze hry

Žluté objekty, které jdou na obrázku 36 vidět, jsou již zmíněné mince, které musí hráč během svého pohybu bludištěm sbírat k získání bodů. Jako poslední funkce, která byla v rámci **Event BeginPlay** volána, byla funkce **Save Location Vector**, která v době kdy byl hráč umístěn do hry spustila zapisování informací o daném hráči do předem připravených polí.

Tato funkce nejdříve zkontroluje zda se opravdu daný hráč nachází na požadované mapě. To má za účelem zamezit vzniku prázdných řádků v souboru, kdy je již **Player Controller** hráče vytvořen, ale hráč ještě nebyl umístěn do hry. Tento rozdíl je sice v řádu milisekund, ovšem jelikož zápis do souboru probíhá 30 krát za sekundu, je tedy možné, že ten to případ nastane. Následně je volána funkce **Set Timer by Event**. Jedná se o funkci UE4, která je schopna dokola spouštět jí přidělenou funkci v mnou zadaných intervalech. Intervalem je v tomto případě hodnota 0.033, která je v sekundách a zajistí mi tak spouštění funkce 30 krát za sekundu.

Funkce, která má být neustále volána během hraní hry je mnou vytvořená **ReadInfo**. V prvním kroku této funkce se do pole **Location Data** uloží aktuální pozice hráče, která je reprezentována třísložkovým vektorem obsahujícím jeho x, y a z souřadnice. Následuje naplnění pole **Player Score Array**, které v každém kroku zavolání této funkce vytáhne z hráčovy struktury jeho skóre a to zapíše na aktuální pozici v poli. Dále zde probíhá zápis do pole typu **Date**, která si ukládá aktuální čas. Další informaci, kterou si funkce ukládá je informace, zda hráč v dané chvíli sebral některý z předmětů nacházejících se v bludišti. Pokud ano uloží do pole **Picked Array** text, který obsahuje slovo „Picked“ a název sebraného předmětu. Následně tuto proměnou nastaví na „notpicked“, aby informace o sebrání byla zapsána pouze jednou a to přesně v tu chvíli, kdy hráč předmět sebral. Důležitou informací pro analýzu chování je také znalost počtu životů daného hráče a zda je hráč v danou chvíli naživu. O uchovávání těchto informací se stará dvojice polí, kde první z nich si v každé iteraci funkce **Set Timer by Event** uloží aktuální počet životů hráče a do pole druhého se mezitím ukládá text v podobě „Alive“ nebo „Dead“ v závislosti na tom, zda byl hráč v danou chvíli živý nebo mrtvý. Poslední

vlastností je zda hráč hodil v danou chvíli bombu. I zde se při každé iteraci funkce přidá hodnota proměnné **Bomb Array Element**, obsahující buďto „Thrown“ nebo „NotThrown“ do pole v závislosti na tom jestli daný hráč během provádění dané iterace bombu hodil nebo ne. Nakonec se hodnota této proměnné nastaví na „NotThrown“, aby stejně jako u sbírání předmětů byl údaj o hození bomby uveden pouze tam, kde má. V nadcházejících podkapitolách bude vysvětleno, jak se hodnoty výše zmíněných proměnných naplňující pole mění během hry.

6.5.1 Implementace házení bomb

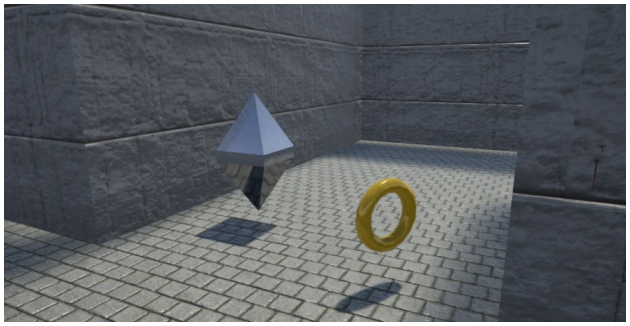
Jako první jsem si musel vytvořit bombu, kterou budou hráči házet. Zvolil jsem jeden ze základních modelů UE4 a to klasickou kouli. Poté jsem si pomocí této koule vytvořil **blueprint**, čímž jsem získal možnost přidávat k bombě různé komponenty a také implementovat její chování. Nejdříve bylo třeba kouli přidat kolizní objekt a nastavit jej tak, aby kolidoval se vším, s čím přijde do kontaktu. Následně jsem přidal komponent **Projectile Movement**, který mé bombě přidal fyzikální vlastnosti střely, čili pokaždé když byla umístěna do scény, čili pokaždé když ji hráč hodil, se chovala, jakoby byla vystřelena.

Dalším krokem byla implementace funkcí, které se staraly o správnou funkčnost bomby. Pokaždé, když se bomba po jejím hození odrazila od země, se zavolala funkce **OnProjectile-Bounce**, která ve svém těle volala funkci **Set Timer by Function Name**, jenž funguje na principu opětovného volání zadané funkce. V mém případě jsem ovšem nepotřeboval tuto funkci plně využívat, jelikož mi stačilo danou funkci zavolat pouze jednou. Tato funkce také jako vstup požadovala čas, po jehož uplynutí zavolá zadanou funkci. Jednoduše se jednalo o čas, za který daná bomba po nárazu o zem vybuchne. Funkce, která se v rámci **Set Timer by Funkcion Name** volala, nejdříve zkontrolovala, zda se v mnou zadané oblasti kolem bomby nachází někdo z hráčů a pokud ano, tak si vzala jeho třídu **Player Controller**, ve které poté zavolala funkci **Take Damage**, která se následně postarala o snížení počtu životů daného hráče. Tato funkce také pokaždé zkontrolovala, zda počet životů neklesl na nebo dokonce pod hodnotu 0. Pokud se tak stalo, tak hráče zabila pomocí funkce **Wake All Rigid Bodies**, což způsobilo, že hráči jakoby zmizela kostra dané postavy a ta následně spadla na zem. Následovala změna proměnné naplňující pole **Health Array** na „Dead“ a poté byl zavolán **respawn** hráče. Ten se v první části staral o opětovné spojení hráčovy kostry s tělem a následně o umístění hráče k jednomu z náhodně vybraných vstupů bludiště. Posledním krokem bylo nastavení výše zmíněné proměnné na „Alive“ a resetování počtu hráčových životů na 100. Po zavolání této série funkcí, starajících se o změnu počtu životů zasaženého hráče, se v rámci **blueprintu** bomby zavolala funkce, která na místě bomby provedla animaci výbuchu a poté bombu zničila.

Co se týče samotného házení bomby, tak měl hráč ve své třídě **Player Controller** naimplementovanou funkci, která danou bombu hodila pokaždé, když zmáčkl pravé tlačítko na myši a zároveň nastavila hodnotu proměnné **Bomb Array Element** na „Thrown“. Nakonec tato funkce spustila odpočet trvající jednu sekundu, který hráči bránil během této doby hodit další bombu.

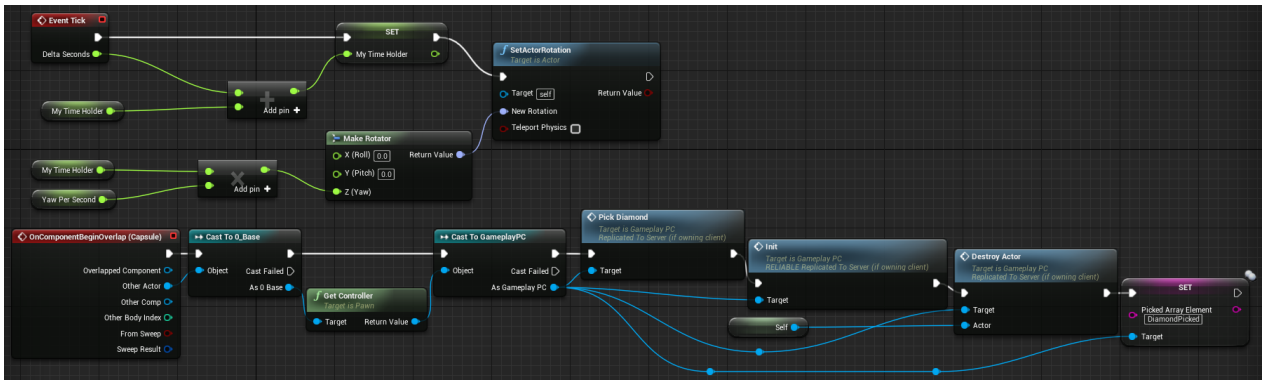
6.5.2 Implementace sběratelných předmětů

Pro potřebu hry bludiště byly vytvořeny dva typy objektů, které se lišili počtem bodů, které hráč obdrží po tom, co je sebere. Jednalo se o zlatý kroužek a diamant. Jelikož je jejich implementace naprosto stejná a liší se pouze v jedné funkci, bude následující popis věnován pouze jednomu ze zmíněných předmětů. Na následujícím obrázku je možné vidět, jak tyto předměty vypadaly.



Obrázek 37: Ukázka sběratelných předmětů

Tyto předměty byly do hry umístěny vždy po načtení příslušné mapy, čili poté co hostující hráč hru spustil. Jednoduchým **for** cyklem se do bludiště umístilo celkem 500 kroužků, respektive se hra pokusila umístit zmíněný počet kroužků, jelikož pokaždé když byl následující kroužek umísťován do hry, se kontrolovalo, zda nebude kolidovat s kroužkem předešlým nebo dokonce se zdmi bludiště a pokud tomu tak bylo, tak umístěn nebyl. Co se týče diamantů, tak ty byly umístěny pouze v okolí středu bludiště a v jejich případě se jich hra pokusila umístit celkem 20. Každý kroužek byl hodnocen jedním bodem a diamant body deseti. Po uplynutí dvou minut hra vždy zkontrolovala aktuální počet obou předmětů a pokud byl menší než mnou zadané číslo tak byl již zmíněný **for** cyklus pro jejich umísťování znovu zavolán, aby se předměty doplnily. Na následujícím obrázku je zobrazena funkce nacházející se v **blueprintu** jednoho z předmětů, starající se o jeho funkčnost.



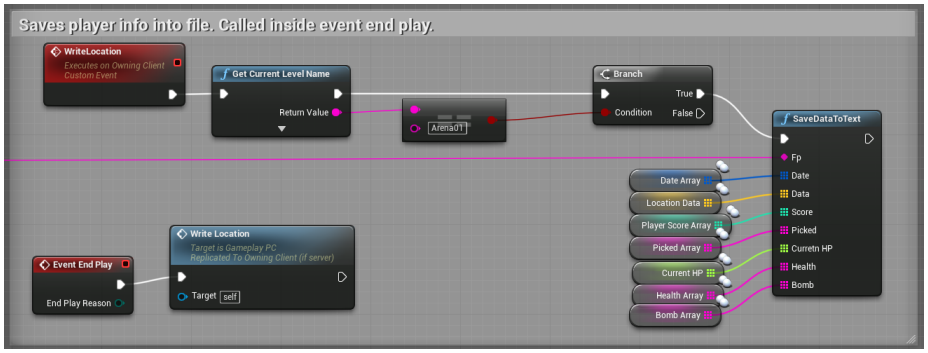
Obrázek 38: Ukázka blueprintu předmětu

Funkce **Event Tick** na obrázku 38 je volána v rámci každého **framu** naší hry. Jejím výstupem je proměnná **Delta Seconds**, která udává čas, který uplynul mezi jednotlivými **framy**. V mém případě se tato funcke stará o vytvoření rotace mých předmětů, aby pouze staticky nestály ve scéně. Každý **frame** se mi uloží daný čas z výstupu funkce, kterým poté vynásobím proměnnou **Yaw Per Second**, pomocí které určuji rychlost rotace a výsledek předám funkci **Make Rotator**, která o danou hodnotu změní rotaci mého předmětu po ose z.

Důležitější funkcí je ovšem funkce druhá a to **OnComponentBeginOverlap**, která je volána vždy, když hráč přijde do kontaktu s daným předmětem. Tato funkce obsahuje spoustu výstupů pro iteraci s kolidujícím objektem, ovšem pro mé účely mi stačil výstup **Other Actor**, který logicky vrací aktora, který přišel do kontaktu s předmětem. Nejdříve jsem zavolal funkci **Cast To**, abych zjistil zda je daný aktor stejného typu jako hráčova postava a poté jsem z dané postavy získal její **Player Controller**, což mi umožňovalo volat z něj funkce, týkající se daného hráče. Jako první byla volána funkce pro změnu počtu bodů, v tomto případě **Pick Diamond**. Funkce byla volána na serveru, jelikož bylo třeba aby ostatní hráči tuto změnu také viděli kvůli aktualizace jejich **widgetu** se skórem všech hráčů připojených do hry. Přesně o to se stará následující funcke **Init**, která jak již bylo zmíněno je i volána po samotném spuštění hry. Dále je volána funkce pro zničení daného předmětu a nakonec změnu proměnné **Picked Array Element** na „DiamondPicked“, aby se sebrání předmětu projevilo také v zápisu do souboru.

6.5.3 Finální zápis do souboru

Když jsem měl tedy všechny proměnné starající se o naplňování příslušných polí nastavené, aby správně zapisovaly hodnoty, mohl jsem přejít k samotnému zápisu. Jelikož UE4 neobsahuje žádné funkce starající se o práci se soubory, musela být tato funkce naimplementovaná pomocí jazyka C++. Jednalo se o jednoduchý zápis do textového souboru, kde vstupem funkce byly veškeré pole zmíněné v kapitole **Implementace herní logiky**. Po kompilaci mého kódu se mi poté daná funkce zpřístupnila v rámci **blueprintů** a mohla být kdekoliv volána. V mém případě jsem volání této funkce provedl, když došlo k ukončení hry a tedy při zavolání funkce **Event End Play**. Na následujícím obrázku je toto volání funkce zobrazeno.



Obrázek 39: Zápis do souboru

Funkce **Write Location** z obrázku 39 nejdříve zkontroluje, zda se hráč nachází na dané mapě, aby nedocházelo ke špatným zápisům a následně zavolá funkci **SaveDataToText**, která byla, jak již bylo zmíněno, naimplementována v jazyce C++. Tato funkce tedy po řádku vypíše hodnoty polí na daném indexu a přiřadí tak danému času danou lokaci, počet bodů, počet životů a stav hráče. Jak lze vidět, tak jejím prvním vstupem je proměnná **Fp**, která udává, kde a pod jakým jménem se má soubor uložit. V mém případě se jméno souboru skládalo vždy s aktuálního data a jména hráče, kterému soubor patřil. Na obrázku níže jde vidět, jak takový soubor vypadá.

```
16:17:33|15.4.2017;-6125.41;-2130.17;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6120.02;-2151.82;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6115.62;-2171.35;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6111.25;-2192.59;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6107.35;-2213;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6103.81;-2232.61;418.2;4;CoinPicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6100.3;-2252.9;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6096.79;-2273.8;418.2;4;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6099.03;-2294.45;418.2;5;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6109;-2321.54;418.2;5;CoinPicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6109;-2321.54;418.2;5;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6118.35;-2340.78;418.2;5;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6128.93;-2359.44;418.2;5;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6134.73;-2379.8;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6137.19;-2403.73;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6137.54;-2422.71;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6136.55;-2444.15;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6134.82;-2464.53;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6132.52;-2485.2;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6129.51;-2508.34;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6126.03;-2532.87;418.2;6;notpicked;100;Alive;NotThrown
16:17:33|15.4.2017;-6126.03;-2532.87;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6123.03;-2553.11;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6119.33;-2578.47;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6116.64;-2598.83;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6114.2;-2620.19;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6112.14;-2641.16;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6110.41;-2661.8;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6109;-2681.71;418.2;6;CoinPicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6107.72;-2704.49;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6106.76;-2728.73;418.2;6;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6106.24;-2749.87;418.2;7;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6106.24;-2749.87;418.2;7;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6106.04;-2776.41;418.2;7;notpicked;100;Alive;Thrown
16:17:34|15.4.2017;-6106.1;-2797.78;418.2;7;notpicked;100;Alive;NotThrown
16:17:34|15.4.2017;-6106.31;-2818.18;418.2;7;notpicked;100;Alive;NotThrown
```

Obrázek 40: Ukázka souboru

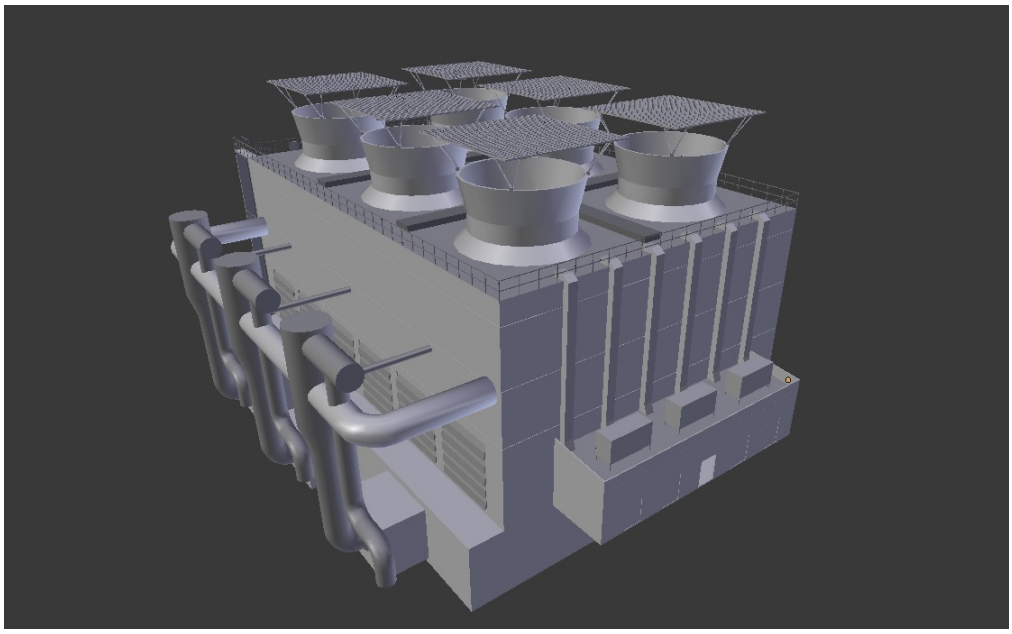
Soubor zobrazený na obrázku 40 patří jednomu z klientů, kteří byli během testování připojení do hry. Zvolil jsem zde ukázku právě klientova souboru, jelikož se v něm vyskytuje jedna chyba, se kterou se při následné implementaci algoritmů zkoumajících chování lidí musí počítat. Nejedná se ani tak o chybu, jako o důsledek odezvy mezi serverem a klientem. Jak je možné vidět na obrázku 40, tak na řádku, kde došlo k sebrání jednoho z předmětů a tedy ke změně skóre hráče, je záznam „CoinPicked“, který udává sebrání zlatého kroužku, zapsán o několik řádků dříve než došlo ke změně skóre hráče. Je tomu tak proto, protože samotná změna proměnné uchovávající text o sebrání předmětu se provádí na straně klienta a tudíž její umístění v souboru odpovídá času a pozici hráče, ovšem změna skóre probíhá na serveru z již zmíněného důvodu, aby o ní věděli všichni hráči a tudíž musí klient počkat až se tato změna na serveru provede a mu se poté pošle nová informace o jeho skóre. Tím se tato změna oproti změně textu o sebrání předmětu zpozdí o zmíněnou odezvu. Veškeré testování hry bludiště a také následné sbírání dat pro analýzu, probíhalo ve spolupráci se studenty VŠB na hodinách předmětu **Modelování v grafických aplikacích**.

7 Vytváření modelu chladící budovy

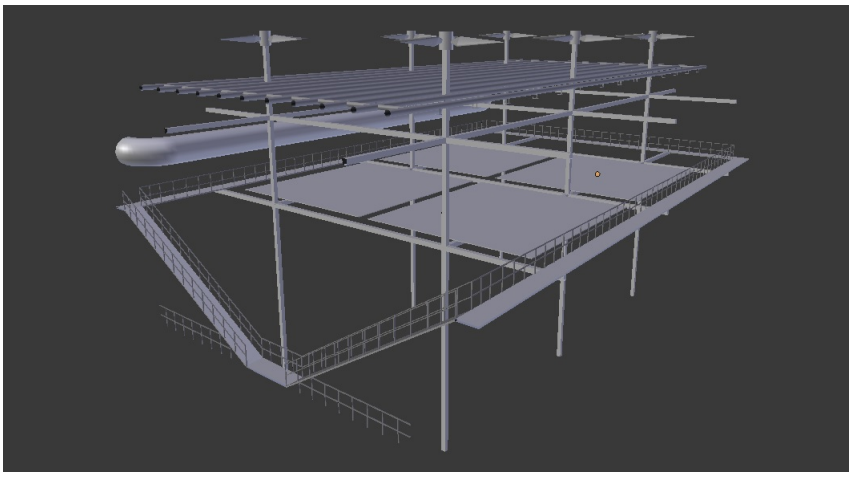
V rámci své diplomové práce jsme byl také součástí skupiny pracující na projektu s názvem **Virtuální prohlídka jaderné elektrárny**. Tento projekt byl vytvářen v prostředí Unreal Engine 4 za pomocí modelovacího nástroje **Bledner**. Projekt byl nejprve vytvořen pro pohled ze třetí osoby a následně upraven pro virtuální realitu. V rámci projektu jsem pracoval na dvou budovách, z nichž jedna byla poté použita v již zmíněné virtuální realitě. Jednalo se o novou chladící budovu nacházející se v jaderné elektrárně Dukovany.

Na rozdíl od klasických chladících věží, které mají 125 metrů, je tato budova sedmkrát nižší a ke chlazení vody využívá mohutné ventilátory. Byla postavena za účelem zvýšení bezpečnosti v elektrárně. Tato budova je na rozdíl od původních chladících věží, schopna odolat zemětřesení a větru, jenž dosahuje rychlosti až 250 km/h. V současné době slouží především k chlazení bezpečnostních systémů. [16]

Model budovy byl rozdělen na vnitřní a venkovní část budovy, jelikož se počítalo s přenášením hráče do vnitřního levelu, jakmile vstoupí do dveří, což mělo za následek snížení náročnosti vykreslování celé scény, jelikož se načítal vždy jen jeden z levelů. Na dvou následujících obrázcích je možné vidět, jak modely jednotlivých částí budovy vypadaly v prostředí **Blender**.

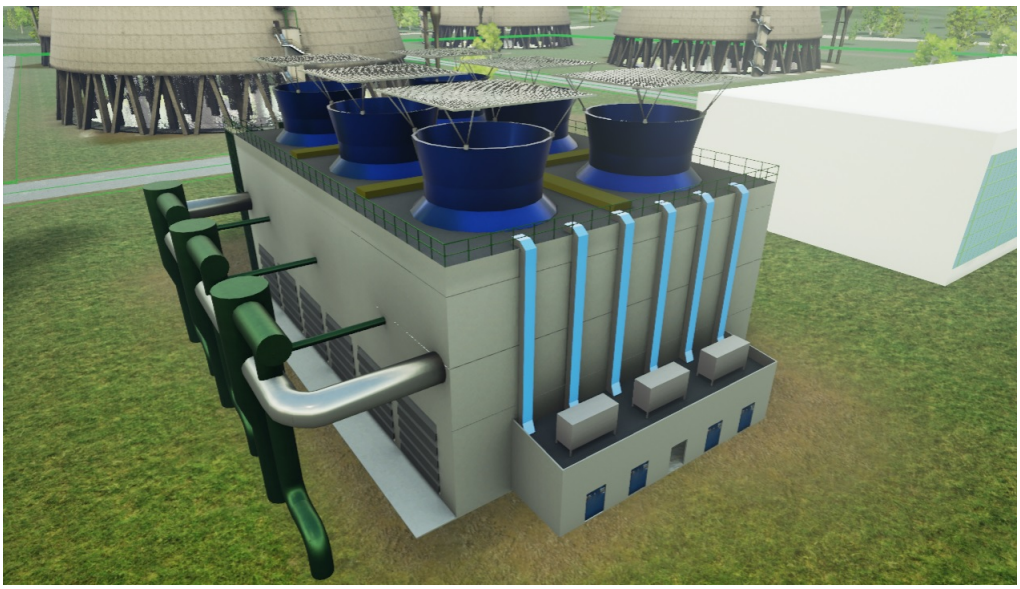


Obrázek 41: Venkovní model chladící budovy



Obrázek 42: Vnitřní model chladící budovy

Modely na obrázcích 41 a 42 byly vytvářeny v poměru 1:1 s reálnou velikostí budovy, čili při importu do prostředí Unreal Enginu 4 se již nemusely provádět žádné úpravy co se týče jejich velikosti. Stejně tomu tak bylo s kolizními objekty, které stejně, jako tomu bylo u modelu bludiště, byly vytvořeny v prostředí **Blender**. Byla zde tedy provedena pouze úprava jejich materiálů, přidáním normálových textur a textury pro ambient occlusion. Následovalo jejich umístění do scény a finální implementace funkcí sloužících pro přenos hráče do budovy z hlavní řídící místnosti pro verzi s virtuální realitou. Jak tato budova vypadala v prostředí Unreal Engine 4 je možné vidět na následujícím obrázku.



Obrázek 43: Chladící budova v prostředí UE4

8 Závěr

V rámci práce byla uvedena a popsána základní teorie týkající se tvorby síťové hry v prostředí Unreal Engine 4. Dále se v práci nachází popis praktické implementace inventáře a systému vyrábění věcí od vytvoření jednotlivých předmětů po jejich zobrazování a používání v prostředí hry více hráčů. V neposlední řadě je zde vysvětlena komunikace mezi jednotlivými hráči pomocí implementace základního chat systému. Následně zde byly získané informace o tvorbě síťové hry využity při vytvoření hry bludiště, která následně sloužila pro zkoumání chování lidí při průchodu bludištěm. Je zde popsáno vytvoření hostování a připojení do hry pomocí takzvané přechodové mapy a způsob ukládání a práce s daty jak na straně serveru, tak na straně klienta. V poslední části práce bylo popsáno vytvoření modelu chladící budovy pro Virtuální prohlídku jaderné elektrárny.

Literatura

[1] Unity [online]. 2017 [cit. 2017-03-04]. Dostupné z: unity3d.com

[2] CryEngine [online]. 2017 [cit. 2017-03-04]. Dostupné z: www.cryengine.com

[3] Unreal Engine [online]. 2017 [cit. 2017-03-04]. Dostupné z: www.unrealengine.com

[4] Networking and Multiplayer. Gameplay Guide [online]. 2017 [cit. 2017-03-04]. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/Networking/index.html

[5] Client-Server Model. Networking and Multiplayer [online]. [cit. 2017-03-13].. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/Networking/Server/index.html

[6] Blueprints Visual Scripting. Unreal Engine 4 Documentation [online]. [cit. 2017-03-13].. Dostupné z: docs.unrealengine.com/latest/INT/Engine/Blueprints/

[7] Gameplay Framework. Gameplay Guide [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/Framework/index.html

[8] Networking and Multiplayer. Multiplayer in Blueprints [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/Networking/Blueprints/index.html

[9] Testing Multiplayer. Gameplay Guide [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/HowTo/Networking/TestMultiplayer/

[10] Widget Blueprints. UMG UI Designer User Guide [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/WidgetBlueprints/

[11] Online Session Nodes. Blueprints Visual Scripting [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/OnlineNodes/

[12] Online Subsystem Overview. Programming Guide [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Programming/Online/index.html

[13] Traveling in Multiplayer. Networking and Multiplayer [online]. [cit. 2017-03-06]. Dostupné z: docs.unrealengine.com/latest/INT/Gameplay/Networking/Travelling/index.html

[14] Casting in Blueprints. Basic Scripting [online]. [cit. 2017-03-30]. Dostupné z: docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/CastNodes/

[15] Blender [cit. 2017-04-09]. Dostupné z: www.blender.org/

[16] Skupina ČEZ spustila v JE Dukovany odolnější chladicí věže za miliardu korun. AtomInfo [online]. [cit. 2017-03-30]. Dostupné z: http://atominfo.cz/2016/03/skupina-cez-spustila-v-je-dukovany-odolnejsi-chladici-veze-za-miliardu-korun/